The Balanced Cube: A Concurrent Data Structure

William J Dally
and
Charles L Seitz

Computer Science
California Institute of Technology

5174:TR:85

The Balanced Cube: A Concurrent Data Structure

William J. Dally
Charles L. Seitz

5174:TR:85

June 20, 1985

(note: revised from February 1985 version)

Abstract

This paper describes the balanced cube, a new data structure for implementing ordered sets. Conventional data structures such as heaps, balanced trees and B-trees have root bottlenecks which limit their potential concurrency and make them unable to take advantage of the computing potential of concurrent machines. The balanced cube achieves greater concurrency by eliminating the root bottleneck; an operation in the balanced cube can be initiated from any node. The throughput of the balanced cube on a concurrent computer is $O(\frac{N}{\log N})$ compared with $O(1)$ for a conventional data structure. Operations on the balanced cube are shown to be deadlock free and consistent with a sequential execution ordered by completion time.

1 Introduction

The appearance of concurrent computers such as the Caltech Cosmic Cube [1] is creating a need for algorithms and data structures which can exploit concurrency. To date these machines have been employed primarily for numerical calculations where the structure of the data is closely matched to the communications structure of the machine. If these machines are to be applied effectively to non-numerical problems, new data structures and algorithms are required.

Sequential computers spend a large fraction of their time manipulating ordered sets of data. For these operations to be performed efficiently on a concurrent computer a new data structure for ordered sets is required. Conventional ordered set data structures such as heaps, balanced trees and B-trees [2] have a single root node through which all operations must pass. This root bottleneck limits the potential concurrency of tree structures making them unable to take advantage of the power of concurrent computers. Their maximum throughput is $O(1)$. This paper proposes a new data structure for implementing ordered sets, the balanced cube, which offers significantly improved concurrency.

The balanced cube eliminates the root bottleneck allowing it to achieve a throughput of $O(\frac{N}{\log N})$. Concurrency in the balanced cube is achieved through uniformity. With the exception of the balancing algorithm, all nodes are equals. An operation may originate at any node and need not pass through a root bottleneck as in a tree structure. In keeping with the spirit of a homogeneous machine, the balanced cube is a homogeneous data structure.

The balanced cube's topology is well matched to binary n-cube multiprocessors. The balanced cube maps members of an ordered set to subcubes of a binary n-cube. A Gray code mapping is used to preserve the linear adjacency of the ordered set in the Hamming distance adjacency of the cube.

The balanced cube algorithms are based on a message passing model of concurrent computation. Each processing node includes a processor and a memory. Nodes must pass messages over an interconnection network to access the memory in other nodes. We assume that the communication time required by this message passing dominates the processing time, which we may safely ignore. This communication cost model accurately reflects the cost/performance behavior of massively concurrent machines constructed from VLSI processing nodes. The Caltech Cosmic Cube is a scale model of such a machine. In this paper we concentrate on developing algorithms which make efficient use of the available communication resources.

In sharp contrast, most existing concurrent algorithms have been developed assuming an ideal shared memory multiprocessor. In the shared memory model, communications cost is ignored. Processes can access any memory location with unit cost and an unlimited number of processes can access a single memory location simultaneously. Performance of algorithms analyzed using the shared memory model does not accurately reflect their performance on large scale multiprocessors.

Previous work on concurrent data structures has concentrated on reducing the interference between concurrent processes accessing a common data base but has not addressed the limited concurrency of existing data structures. Kung and Lehman [3] have developed

concurrent algorithms for manipulating binary search trees. Lehman and Yao [4] have extended these concepts and applied them to B-trees. Algorithms for concurrent search and insertion of data in AVL-trees [5] and 2-3 trees [6] have been developed by Ellis.

These papers introduce a number of useful concepts that minimize locking of records, postpone operations to be performed, and use marking mechanisms to modify the data structure. However, these papers consider the processes and the data to be stationary, and thus do not address the problems of moving processes and data between the nodes of a concurrent computer. The cost of communications, which we assume to dominate processing costs, has largely been ignored.

The remainder of this paper describes the balanced cube and how it addresses the issues of correctness, concurrency, and throughput. In the next section the data structure is presented and the consistency conditions are described. This section also discusses the assumptions which are made in analyzing the balanced cube, and the environment in which the data structure and its algorithms are expected to reside. The VW search algorithm is described in Section 3. VW search uses the distance properties of the Gray code to search the balanced cube for a data record in $O(\log N)$ time while locking only a single node at a time. An insert algorithm is presented in Section 4. Insertion is performed by recursively splitting subcubes of the balanced cube. Section 5 discusses the delete operation. Deletion is accomplished by simply marking a record as deleted. A background garbage collection process reclaims deleted subcubes. The insertion and deletion algorithms tend to unbalance the cube. A balancing algorithm, presented in Section 6, acts to restore balance. Each of the algorithms presented in this paper is analyzed in terms of complexity, concurrency, and correctness. Section 7 extends the balanced cube concept to B-cubes which store several records in each node. Finally, Section 8 discusses the results of experiments run to verify the balanced cube algorithms.

2 Data Structure

2.1 The Ordered Set

An *ordered set* is a set, S, of objects on which a linear ordering $<$ has been defined, $\forall a, b \in S$ either $a < b$ or $b < a$ and $a \neq b$ unless a and b are the same object. In many applications these objects are records and the linear order is defined by the value of a key field in each record. In this context the ordered set is used to store a database of relations associating the key field with the other fields of the record. The order relation defined on the keys of the records is implicit in the structure. A data structure implementing the ordered set must efficiently support the following operations.

search(key): return the object associated with a key.

insert(object): add an object to the set

delete(key): remove the object associated with key from the set.

search(lkey,ukey): return the set of objects with keys in the range [lkey,ukey].

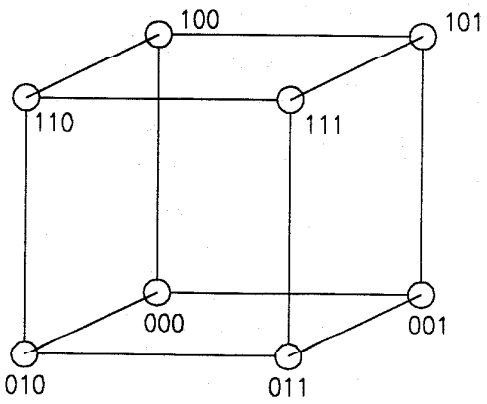

Figure 1: Binary 3-Cube

succ(key1): return the object with the smallest key greater than key1.

pred(key1): return the object with the largest key smaller than key1.

max: return the maximum object.

min: return the minimum object.

In this paper we will restrict our attention to developing algorithms for the search, insert and delete operations. The remaining functions can be implemented as simple extensions of these three fundamental operations. The succ and pred functions can be implemented using the nearest neighbor links present in the balanced cube.

2.2 The Binary n-Cube

The balanced cube is a data structure for representing ordered sets that stores data in subcubes of a binary n-cube [7]. A binary n-cube has $N = 2^n$ nodes accessed by n-bit addresses. Each bit of the address corresponds to a dimension of the cube. In this paper the node or subcube with address a_i is referred to as $N[a_i]$. If the address is implicit, the node will be referred to as N. The binary n-cube is connected so that node $N[a_i]$ is adjacent to all nodes whose addresses differ from a_i in exactly one bit position: $\{a_i \oplus 2^j \mid 0 \leq j \leq n-1\}$. A binary 3-cube with nodes labeled by address is shown in Figure 1.

An m-subcube of a binary n-cube is a set of $M = 2^m$ nodes whose addresses are identical in all but m positions. An m-subcube is identified by an address which contains unknowns, represented by the character X, in the m bit positions in which its members addresses may differ. For example in Figure 1 the top of the 3-cube is the 1XX subcube. The top right edge is the 1X1 subcube.

A right m-subcube is an m-subcube which has unknowns in the least significant m bits of the address. No X is to the left of a 0 or 1 in a right subcube address. For example, the 1XX subcube is a right subcube while the 1X1 subcube is not. A node is a right 0-subcube since it has zero Xs in its address. We can think of a node as a singleton set. The *corner node* of a right subcube $N[a]$ is the node with the lowest address in the subcube, $N[f(a)]$ where f is defined as

$$f(a) = a_n \mid a_n \in a,\ a_n \leq a_j\ \forall a_j \in a. \tag{1}$$

The corner node address is the subcube address with all unknown bits set to zero. The *upper nodes* of a right subcube $N[a]$ are all the nodes in the subcube other than the corner node: the elements of the set $N[a] \setminus N[f(a)]$.

2.3 The Gray Code

The balanced cube uses a Gray code [8] to map the elements of an ordered set to the vertices of a binary n-cube. Consider an integer, I, encoded as a weighted binary vector, $b_{n-1}, \ldots, b_0$, so that

$$I = \sum_{j=0}^{n-1} b_j 2^j. \tag{2}$$

The reflected binary code or Gray code representation of I, a bit vector $G(I) = g_{n-1}, \ldots, g_0$, is generated by taking the modulo-2 sum of adjacent bits of the binary encoding for I [8].

$$g_i = \begin{cases} b_i \oplus b_{i+1} & \text{if } i < n-1 \\ b_i & \text{if } i = n-1 \end{cases} \tag{3}$$

Since the $\oplus$ operation is linear, we can convert back to binary by swapping g_i and b_i in equation 3. We use the function $B(J)$ to represent the binary number whose Gray code representation is J.

$$b_i = \begin{cases} g_i \oplus b_{i+1} & \text{if } i < n-1 \\ g_i & \text{if } i = n-1 \end{cases} \tag{4}$$

By repeated substitution of equation 4 into itself we can express b_i as a modulo-2 summation of the bits of $G(I)$.

$$b_i = \sum_{j=i}^{n-1} g_j \pmod 2. \tag{5}$$

While these equations serve as a useful recipe for converting between binary and Gray codes, we gain more insight into the structure of the code by considering a recursive list definition of the Gray code. For any integer, n, we can construct a list of $N = 2^n$ integers, gray(n) so that the I^{th} element of gray(n) is an integer whose binary encoding is identical to the Gray encoding of I. The construction begins with the Gray code of length 1. At the i^{th} step we double the length of the code by appending to the current list a reversed copy of itself with the i^{th} bit set to one.

$$\text{gray(0)} = [0]. \tag{6}$$

$$\text{gray(n)} = \text{cat(gray(n} - 1), 2^{(n-1)} + \text{rev(gray(n} - 1))). \tag{7}$$

It is this reversal that gives the code the symmetry and reflection properties that we will use in developing the balanced cube search algorithm.

The Gray code mapping preserves adjacency. In the linear space of the ordered set, element I is adjacent to elements $I \pm 1$. In the cube space, however, the distance between two nodes is the Hamming distance between the node addresses: the number of bit positions in which the two addresses differ. For nodes A and B to be adjacent, they must be Hamming distance one apart, $d_H(A, B) = 1$. The hamming distance between I and $I - 1$, $d_{HA}(I)$ is given by the recursive equation.

$$d_{HA}(I) = \begin{cases} d_{HA}(\frac{I}{2}) + 1 & \text{if } 2|I,\ I \neq 0 \\ 1 & \text{if } 2 \not| I \\ \text{undefined} & \text{if } I = 0 \end{cases} \tag{8}$$

For example, in the case where $I = \frac{N}{2}$ and $I-1 = \frac{N}{2} - 1$ the elements are at opposite corners of the cube, distance n apart. The Gray code has the property that $d_H(G(I), G(I+1)) = 1$, $\forall I \ni 0 \leq I \leq (N-2)$. Thus, if we map element I of the linear order to node $G(I)$ of the binary n-cube, nodes that are adjacent in linear space are also adjacent in cube space.

2.4 The Balanced Cube

In a balanced cube, each datum is associated with a right subcube, $N[a_i]$ and is stored in the corner node of the subcube $N = N[f(a_i)]$. A datum is composed of a key, $N.key$, a record, $N.record$, the dimension of the subcube, $N.dim$, and a flag to indicate if the subcube is deleted or free $N.flag$. The data is ordered so that if $\text{B}(a_1) > B(a_2)$, $N[a_1].key \geq N[a_2].key$. Node addresses are ordered using the inverse Gray code function; thus if two addresses are adjacent in the order they will also be Hamming distance one apart.

Upper nodes of the subcube $N[a_i]$ are flagged as slaves to the corner node by setting $N.flag$ = SLAVE. Any messages transmitted to an upper node $N[a_u]$ are routed to the corner node of the subcube to which $N[a_u]$ belongs. There is one exception to this routing rule. A split message is always accepted by its destination and never forwarded. This message is the mechanism by which upper nodes become corner nodes. Since the cube is balanced most corner nodes have dimensions differing only by a small constant. Thus, the message routing time between adjacent corner nodes will be limited by a small constant.

Data are associated with the subcubes rather than the nodes of a binary n-cube to allow ordered sets of varying sizes to mapped to a cube of size 2^n. For example, a singleton set mapped to the 3-cube of Figure 1 would be associated with the subcube XXX, the entire cube. If a second element is added to the set, the cube will be split. One element will be associated with the $0XX$ subcube and the other element with the $1XX$ subcube. This splitting is repeated as more elements are added to the set.

A balanced cube is balanced in the sense that in the steady state, the dimensions of any two subcubes of the balanced cube will differ by no more than one. This degree of balance guarantees $O(\log N)$ access time to any datum stored in the cube. The balance condition is valid only in the steady state. Several insert or delete operations in quick succession may unbalance the cube. A balancing process which runs continuously acts to rebalance the cube.

There are two consistency conditions for a balanced cube. It must be ordered as described above and operations on the cube must be serializable giving results consistent with sequential execution of the same operations ordered by time of completion. Note that unordered data may pass through nodes of the cube during an operation; however when those data are stored, they must be stored in a manner which preserves ordering. The second consistency condition guarantees correct results from concurrent operations. A group of concurrent operations must give the same result as the same operations executed sequentially in order of completion. Operations which complete simultaneously must give the same result as any ordering of the operations.

When designing algorithms for concurrent computers it is important that the processing costs and the communications costs be balanced. This balance can be achieved by varying the *grain size* or *granularity* of the algorithm: the amount of computation performed at a single processing node. A fine grain computation performs very little work and stores very little information at each processing node; a large grain computation performs a large amount of work and stores a large amount of information at each node. In concurrent machines with a limited amount of memory at each processing node it is especially important that the grain size be selected to make efficient use of all available memory. The B-cube data structure is an extension of the balanced cube structure which allows the grain size to smoothly varied from storing one datum per processing node to storing the entire data structure in one node. This flexibility in selecting grain size allows the data structure to be implemented on a wide range of concurrent machines.

2.5 Environment

The data structure and algorithms described in this paper are intended for implementation

on a message passing concurrent computer. Such a computer consists of a number of processors connected by communications channels. The computer is assumed to support a binary n-cube communications topology. It may be physically connected as a binary n-cube, or have embedded a binary n-cube topology in another network. The Caltech Cosmic Cube is an example of a computer which is physically connected as a binary n-cube [1]. For purposes of analysis, it is assumed that passing a message between two processors adjacent in the cube requires unit time. The cube algorithms are implemented by sequential processes running on each of the processors which communicate by means of messages.

The discussion of the cube algorithms distinguishes between requester objects which request service from the cube, and node objects which implement the cube. No assumptions are made about the location of requester objects in the computer. Requesters may share processors with nodes or they may reside on separate processors.

All of the balanced cube algorithms are described in terms of messages. A message is transmitted either from a requester to a node of the cube to initiate a cube operation, or from one node of the cube to a second node of the cube to continue an operation. Message transmission is indicated in the algorithms by the statement `send(<destination>,<message type>(<arguments>),<wait>)`. This statement causes the message to be transmitted to the destination node. If the wait argument is true (`WAIT`), the sender waits for a reply, otherwise (`NOWAIT`) the sender continues execution without waiting for a reply. Message transmission over a single link takes unit time. It is assumed that message transmission time dominates execution time for the algorithms. Except for causality, the algorithms make no assumptions about the order in which messages are received. When a message is received by a node it initiates a procedure which takes the message contents as arguments and begins executing on the destination node. In each of these procedures, the address of the present node is implicit so all references to the present node record are listed as $N.<field>$.

Some of the procedures initiated by messages lock the node they are executing on. Most of these locks, described by the statements `LOCK` and `UNLOCK`, are write-locks. They prevent other messages from locking the node or modifying its state; however messages that only read a node may proceed even if an node is write-locked. Write-locks do not create a critical region, and a snapshot may occur between two write-locks. Since most procedures lock only a single node and do not wait for replies from any locking messages before they unlock the node, deadlock is impossible. Read locks are used by some procedures to implement a simultaneous update of two fields (such as key and record). Whenever a read lock is invoked a writelock is implied as well. To assure that deadlock will not occur, special care is taken whenever the code in the critical region, between the statements `READLOCK` and `READUNLOCK`, waits for a reply from any message. For example, the garbage collector sends the mergedown message with the wait field set in a critical region. In this case a total ordering of the nodes in the cube is used to resolve conflicts so a cycle of conflicts, and hence deadlock, is impossible.

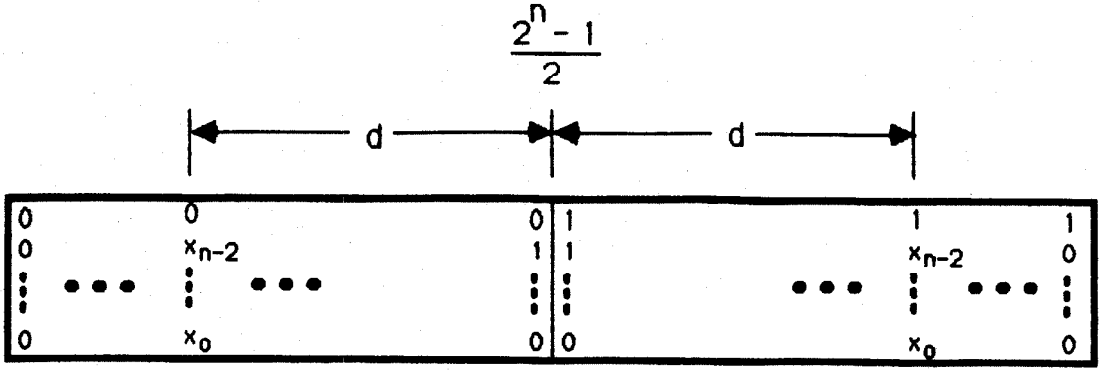

Figure 2: Calculating Distance by Reflection

3 Search

3.1 Distance Properties of the Gray Code

In order to develop a search algorithm for the balanced cube we need to know the distance properties of the Gray code. That is for any element of the ordered set mapped onto the cube, at what distance in linear space are its neighbors in cube space The distance properties of the mapping tell us how much we can reduce the (linear) search space with each nearest neighbor query in the cube. To achieve $O(\log N)$ search time we must cut the search space in half with no more than a constant number of messages.

The reflection properties of the Gray code give us an easy method of calculating distance in a balanced cube. Consider some node, X, in a balanced n-cube. As shown in Figure 2, if we toggle the most significant bit of node address X, we generate address $Y = X \oplus 2^{n-1}$. In linear space, Y is the reflection of X through $\frac{2^n-1}{2}$. Thus the linear distance between node X and its neighbor, Y, in the $n-1^{st}$ dimension is

$$d_{LN}(X, n-1) = 2\left|X - \frac{2^n - 1}{2}\right|. \tag{9}$$

To calculate the distance in a lower dimension, say k, we reflect about the center of the local gray(k) list. Thus, the linear distance from a node with address X to its neighbor in the k^{th} dimension is given by

$$d_{LN}(X, k) = 2\left|(X(\text{mod } 2^{k+1})) - \frac{2^k}{2}\right|. \tag{10}$$

Figure 3 shows the distance function for a balanced 5-cube. The symmetry of reflection is clearly visible. In each dimension, k, we have $2^{(n-1)-k}$ Vs centered on sublists of dimension k. We use these Vs in the following section to develop a new search algorithm.

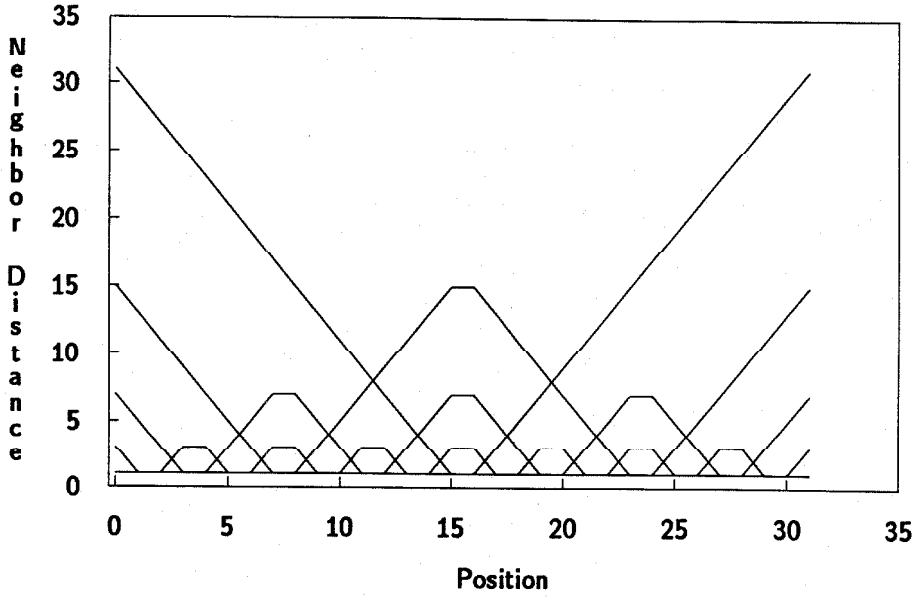

Figure 3: Neighbor Distance in a Gray 5-Cube

3.2 VW Search

VW search attempts to find a search key in the Gray cube by traversing the Vs and Ws of the distance function shown in Figure 3. The neighbors of a node, X, are those nodes that are directly across a V from X in Figure 3. The search procedure sends messages across these valleys selecting a search path that guarantees that the search space is halved every two messages.

Messages:

VW search is performed by passing messages between the nodes of the cube being searched. The body of the search uses a single message: VWsearch. When a node receives a VWsearch message it updates the state fields of the message and forwards it to the next node in the search path. Nodes never wait for a reply from a message. The format of the search message is shown below. The search state is represented by the destination node, two dimensions: `Vdim` and `Wdim`, and a search mode.

VWSEARCH	Requester	Key	Node Address	VDim	Wdim	Mode

In VW search we encode the search space into the destination address, `self`, and a dimension, `Wdim`. `Wdim` is the dimension of the smallest W (two adjacent Vs) in the distance function which contains the search space. A second dimension `VDim` is the dimension of the

smallest V which completely contains our current W and thus the search space. `Vdim` can be computed from `Wdim` and `self`; however it is more convenient to pass it in the message than to recompute it at each node.

The `Wdim, self` encoding of the search space can be converted to the conventional upper bound, lower bound (U, L) representation by means of the reflect function. The reflection in the linear space about dimension, d, of node X is given by

$$f_R(X,d) = X - 2(X(\text{mod } 2^d)) + 2^d - 1. \tag{11}$$

The current position, `self` or its reflection in the `Wdim` dimension is one bound of the search space, and the reflection of this bound in the `Vdim` dimension is the other bound. Thus, if the current address is S, the `Wdim` is W, and the `Vdim` is V, we can calculate the linear bounds of the search space (L,U) from

$$L(S,W,V) = \min(S, f_R(S,W), f_R(S,V), f_R(f_R(S,W),V)). \tag{12}$$

$$U(S,W,V) = \max(S, f_R(S,W), f_R(S,V), f_R(f_R(S,W),V)). \tag{13}$$

The `Mode` field of the search message encodes how much of the current W has been searched. In W mode, the search key may be in either trough of the W, thus the W search procedure may have to reflect the search to the other trough of the W. In V mode, the search key must be in the current trough or V of the W. Since there is no need to reflect to the other side of the W, the dimension `Wdim` will always decrease in V mode.

When search terminates unsuccessfully, a query message is sent to the linear address neighbor of the current node to verify that the search key has not been inserted while the search was underway. The linear address neighbor replies with the value of its key. If the current key and the neighbor's key bracket the search key then the search completes by sending a `NOTFOUND` reply to the requester. Otherwise, the search continues. The formats of the `QUERY` and `REPLY` messages are shown below.

QUERY	Requester	Node Address

REPLY	Node Address	Key

Algorithm:

The code for VWSearch is given below in a pseudocode modeled after C [9].

```
/*-------------------------------------------------------------*/
/*
```

```
 *    VWSearch(Requester, key, Vdim, Wdim, Mode)
 *    Searches a balanced cube for key.  When the key is found a
 *    reply message is sent to the Requester.  Vdim, Wdim and
 *    Mode encode the state of the search.
 */
/*-------------------------------------------------------------*/
VWSearch(Requester,key,Vdim,Wdim,Mode)
{
  LOCK ;
  switch(Mode) {
    case V: VSearch(Requester,key,Vdim,Wdim) ; break ;
    case W: WSearch(Requester,key,Vdim,Wdim) ; break ;
  }
  UNLOCK ;
}
/*-------------------------------------------------------------*/
/*
 *  If the key is between the current node and its neighbor,
 *  the V dimension is set to the W dimension and the W dimension
 *  is reduced to prevent searching back to the previously visited
 *  neighbor.
 *
 *  In any case, the W dimension is reduced until a neighbor in
 *  the proper direction is found and this neighbor is searched.
 */
/*-------------------------------------------------------------*/
VSearch(Requester,key,Vdim,Wdim)
{
  if(KeySameSideAsNeighbor(self,key,self.key,Wdim)) {
    Vdim = Wdim ;
    Wdim = Wdim - 1 ;
  }
  Found,Wdim = ReduceDimension(self,self.key,key,Wdim) ;
  if(Found) send(Requestor,Reply(FOUND),NOWAIT) ;
  if(Wdim < N.dim) CheckNotFound(key) ;
  else
    send(Neighbor(self,Wdim),
         VWSearch(Requestor,key,Vdim,Wdim,W),NOWAIT) ;
}

/*-------------------------------------------------------------*/
/*
 *  If the key is on the same side as our neighbor (in this V), then
 *  the search proceeds as above in Vsearch
 *
 *  Otherwise, the search is reflected across the V dimension to the
 *  other side of the W.
 */
```

```
/*-------------------------------------------------------------*/
WSearch(Requestor,key,Vdim,Wdim)
{
  if (KeySameSideAsNeighbor) {
    VSearch(Requestor,key,Vdim,Wdim) ;
  }
  else {
    Found,Vdim = ReduceDimension(self,self.key,key,Vdim) ;
    if(Found) send(Requestor,Reply(FOUND),NOWAIT) ;
    if(Vdim < 0) CheckNotFound(key) ;
    else
      send(Neighbor(self,Vdim),
      VWSearch(Requestor,key,Vdim,Wdim,V),NOWAIT) ;
  }
}

/*-------------------------------------------------------------*/
/*
 *    ReduceDimension(addr, key1, key2, dim)
 *    Finds the largest dim, less than or equal to the present
 *    value for which addr has a neighbor in the direction of
 *    the search key (key2). Returns found if the keys are equal
 *    and returns a negative dimension if there is no neighbor
 *    in the appropriate direction.
 */
/*-------------------------------------------------------------*/
ReduceDimension(addr, key1, key2, dim)
{
  dim = min(dim,n-1) ;
  if (key1 == key2) return(TRUE,dim) ;
  while (!(KeySameSideAsNeighbor(self, key1, key2, dim)) &&
         (dim >= 0)) dim-- ;
  return(FALSE,dim) ;
}
/*-------------------------------------------------------------*/
/*
 *    CheckNotFound(key) - Checks if current node and its
 *    linear neighbor bracket the search key.  Details of
 *    Lin_Neighbor and bracket have been omitted.
 */
/*-------------------------------------------------------------*/
CheckNotFound(key)
{
  key1 = send(Lin_Neighbor(self,self.key,key),Query(),WAIT) ;
  if(bracket(self.key,key,key1)) send(Requestor,Reply(NotFound)) ;
  send(Lin_Neighbor(self,self.key,key),
       VWSearch(Requestor,key,N.dim+1,N.dim+1,V),NOWAIT) ;
}
```

```
/*-------------------------------------------------------------*/
/*
 *    Tests if the neighbor of addr is on the same side of addr
 *    as key2 is of key1.
 */
/*-------------------------------------------------------------*/
KeySameSideAsNeighbor(addr, key1, key2, dim)
{
  return(
    ((key1 > key2) && (ToBinary(addr) > ToBinary(Neighbor(addr,dim)))) ||
    ((key1 < key2) && (ToBinary(addr) < ToBinary(Neighbor(addr,dim))))
  )
}

/*-------------------------------------------------------------*/
/*
 *      Computes the neighbor of addr in the dim-th dimension
 */
/*-------------------------------------------------------------*/
Neighbor(addr,dim)
{
  return(addr^(1<<dim)) ;
}
/*-------------------------------------------------------------*/
/*
 *    converts Gray code to binary
 */
/*-------------------------------------------------------------*/
ToBinary(i)
{
  j = i ;
  for(k = 1;k < n;k++) j = j^i>>k ;
  return(j) ;
}
/*-------------------------------------------------------------*/
```

Example 3.1 The search technique is best described by means of an example. Figure 4 shows the search of a Gray 4-cube for the key stored at node $G(6)$. The search begins at node $G(2)$. The search is started with the message VWSearch(6,5,5,V). Since we know that the search key must be in the current dimension 5 trough of the W (this is the whole 4-cube), we start the search in V mode. The subsequent search messages are as follows:

1. Node $G(2)$ sends the message VWSearch(6,5,4,W) to its dimension 4 neighbor, $G(13)$.
2. Since the key is between $G(2)$ and $G(13)$, $G(13)$ sends the message VWSearch(6,4,3,W) to node $G(10)$.
3. The search key is not between $G(13)$ and $G(10)$, so $G(10)$ must reflect the search (in the

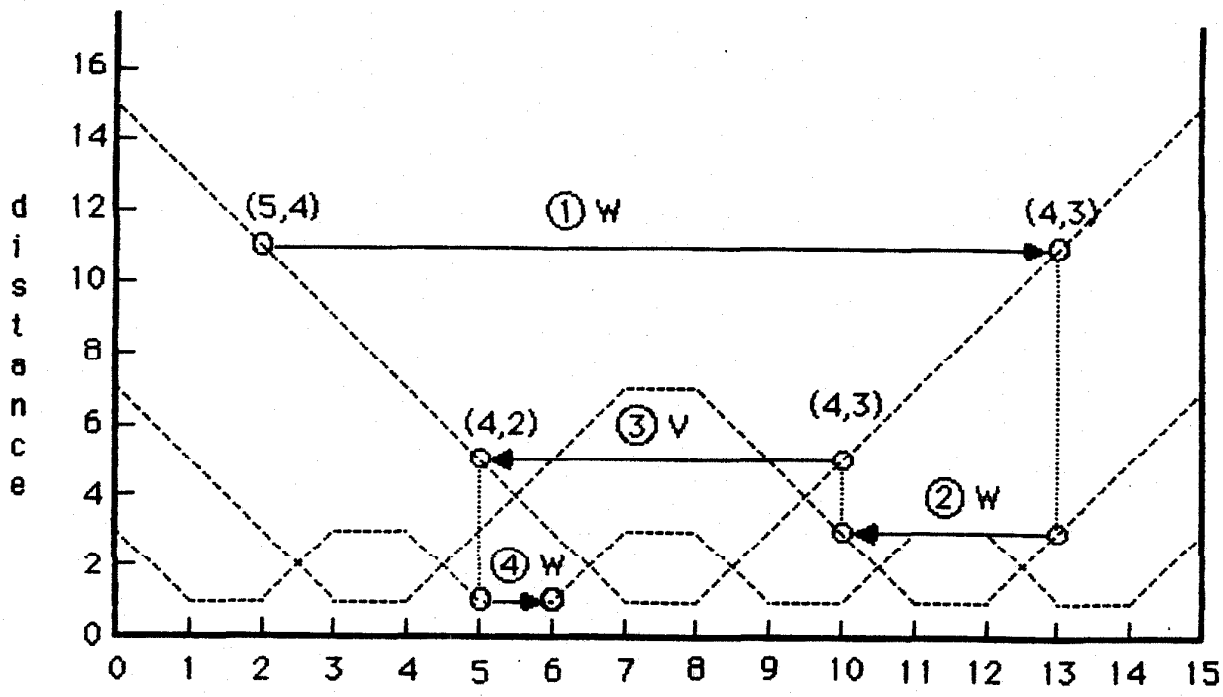

Figure 4: Example of VW Search

Vdim dimension) to the other trough of the W by sending the message VWSearch(6,4,3,V) to node $G(5)$.

4. Since the key is not between $G(5)$ and its neighbor, $G(2)$, the Wdim is decreased to find a neighbor in the direction of the key, $G(6)$. $G(5)$ sends the message VWsearch(6,4,2,W) onto $G(6)$ where the search terminates.

Example 3.2 Figure 5 shows two examples of searching a balanced cube which is not full and which is temporarily out of balance.

1. In the first example, a search for the contents of $G(4)$ is initiated from node $G(5)$. By the message VWSearch(4,4,4,V).

2. Since the search key is less than $G(5)$, node $G(5)$ forwards the search message to node $G(2)$, a slave node of node $G(0)$ with the message VWSearch(4,4,3,W). Since the search key is greater than the contents of $G(0)$ the W dimension is decremented to 0 and the message VWSearch(4,4,0,W) is sent to node $G(3)$. It is important to note that although $G(2)$ is a slave node, and thus uses the value of the corner node, $G(0)$, the search continues from $G(2)$ and is not detoured to $G(0)$.

3. Node $G(3)$ is also a slave of $G(0)$ and thus less than the search key, so the search is reflected across the V dimension to node $G(4)$ with the message VWSearch(4,4,0,V). The search key is found at node $G(4)$.

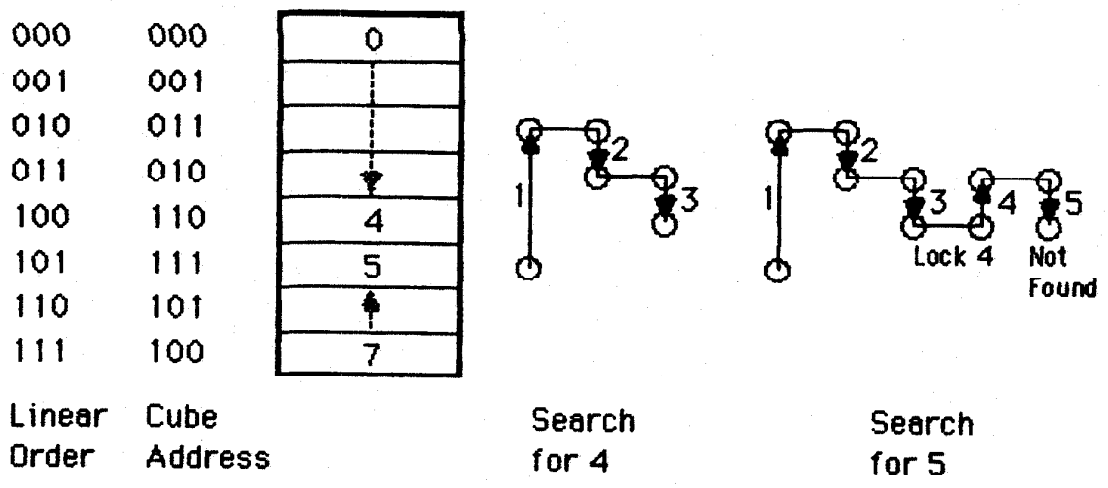

Figure 5: VW Search Example 2

The second example illustrates the case when the search key is not present in the cube.

1-3. The search for the key 3, is initiated at node $G(5)$. The search proceeds as above until the message VWSearch(4,4,0,V) reaches node $G(4)$.

4. To confirm that the key has not been inserted during the search, node $G(4)$ sends a `QUERY` message to its linear address neighbor, node $G(3)$.

5. Node $G(3)$ replies with the value associated with its subcube, 0. Since 0 and the contents of $G(4)$, 4, bracket the search key the search terminates by sending a not found reply to the requesting process.

The remainder of this section analyzes the VW search algorithm to show that the order of the algorithm is $O(\log N)$ and to prove that the algorithm is deadlock free.

Lemma 1 Each execution of `VSearch` decreases `Wdim` by at least 1.

Proof: There are two cases:

1. If the search key is between the current node, `self`, and its neighbor in `Wdim` we explicitly decrement `Wdim`.

2. If the search key is not between `self` and `Neighbor(self,Wdim)` then the first test in `ReduceDimension` will fail and `Wdim` will be decremented by `ReduceDimension`.

■

Lemma 2 `VSearch` is executed at least once for every two search messages.

Proof: Each search message causes either `WSearch` or `VSearch` to be executed. The only case where `VSearch` is not executed is when the `Mode` field of the message is `W` and the key is not between the current node and its neighbor. The next message generated in this case has a `Mode` of `V` which will cause `VSearch` to be executed. Thus `VSearch` will be executed at least once for every two search messages. ■

Theorem 1 A VW Search of a Gray n-cube requires no more than $2(\log N + 1)$ messages.

Proof: From Lemmas 1 and 2 `Wdim` is decremented at least once every two messages. Since `Wdim` is initially $n = \log N$, after $2 \log N$ messages `Wdim` will be zero. An additional two messages will either find the search key or decrement `Wdim` below zero causing termination. ■

Theorem 2 The VW Search algorithm is deadlock free.

Proof: The VW Search algorithm locks only one node at a time: the one currently conducting the search. Since no read locks are placed, the `QUERY` messages are never blocked. Thus, there is no possibility of deadlock. ■

4 Insert

Messages:

The insert operation is initiated by sending an insert message to any node in the cube. The insert message contains the address of the process initiating the search, the destination node, the bounds of the search space, and the key and record to be inserted. When the insert is completed, a reply message is transmitted to the requesting process to signal successful or unsuccessful completion. A split message is used by the inserting process to split an existing right subcube into two right subcubes of lower dimension to make room for the insert.

INSERT	Requester	Node Address	Bounds	Key	Record

SPLIT	Node Address	Dimension	Key	Record

Algorithm:

The insert algorithm is identical to the search algorithm except that on completion, in addition to sending a reply, the insert splits a node and inserts the key and record. Rather than repeat the search algorithm here, only the changes will be described.

If the key being inserted is already in the cube, the insert terminates with a found message since the same key cannot be stored at two different locations in the cube. If the key being inserted is not already in the cube, the insert procedure must insert it. To do this, the not found reply of the search procedure listed above:

```
        send(Requestor,Reply(NotFound)) ;
```

is replaced by a call to the the following insert procedure:

```
/*-------------------------------------------------------------*/
/*
 *      insert(addr,key,record) - Inserts key, record
 */
/*-------------------------------------------------------------*/
insert(addr,key,record) ;
{
  if (N.Dim > 0) {
    N.Dim-- ;
    neighboraddress = self XOR 2^N.Dim ;
    if (((Key > N.key) && (B(neighboraddress) > B(self))) ||
        ((Key < N.key) && (B(neighboraddress) < B(self)))) {
      send(neighboraddress,
           SPLIT(Requester,Dimension,Key,Record),NOWAIT) ;
    }
    else {
      READLOCK
      tempKey = N.key ;
      N.key = key ;
      tempRecord = N.record ;
      N.record = record ;
      send(neighboraddress,
           SPLIT(Requester,Dimension,tempKey,tempRecord),WAIT) ;
      READUNLOCK
    }
    N.dim-- ;
    send(Requester,REPLY(INSERTED),NOWAIT) ;
  }
  else {
    send(Requester,REPLY(FULL),NOWAIT) ;
  }
}
```

```
/*-------------------------------------------------------------*/
/*
 *      Split is invoked by receipt of a SPLIT message.
 */
/*-------------------------------------------------------------*/
Split(Requester,Address,Dimension,Key,Record)
{
  READLOCK ;
  N.key = Key ;
  N.record = Record ;
  N.dim = Dimension ;
  READUNLOCK ;
}
/*-------------------------------------------------------------*/
```

If the present node has a dimension greater than zero, then it is split by sending a split message to its upper half and decrementing its dimension. If the dimension is already zero, the insert terminates with a reply of FULL. This does not necessarily mean that the cube is full. The cube may just be temporarily out of balance.

If the insert key and the linear order of the neighbor's address have the same relation to the current key and current address, the split message inserts the key and record into the corner node of the upper half subcube and sets its dimension to prevent it from routing further messages to the original corner node. When the dimension is set, the split is complete in that the split node will begin responding to messages rather than routing them to the corner node.

If the insert key and the linear order of the neighbor's address have opposite relations to the current key and current address, the split message copies the original corner node's key and record into the upper half subcube. The lower half subcube is then set with the new key and record. Note that between the assignment of the key and the assignment of the record to the lower half subcube this subcube is in an inconsistent state. Thus, the node is read locked during the entire copy and update procedure. This locking cannot cause deadlock since the split node is in fact part of the locked node until the split is completed. This is an important distinction.

To prevent the possibility of simultaneously inserting the same key in the cube twice, it is necessary that the search terminate in the UP direction unless the insert key is lower than the lowest key in the cube.

Consider splitting the subcube 000XXX into 0000XX and 0001XX. In the instant of time before the split, all nodes in 000XXX must route their messages to 000000. Immediately after the split, all messages to the upper half subcube 0001XX must be routed to 000100. For the cube algorithms to operate correctly, the split must be an atomic operation. Since the split occurs when the node's dimension is written, it is an indivisible operation. Before the dimension is written, messages to nodes 0001XX are routed to 000100 which forwards them to 00000 since it is not a corner node. After the dimension is written these messages are accepted directly by 000100. Because the key and record of the split node are in fact not accessible before the dimension is updated readlocks on the split procedure are not

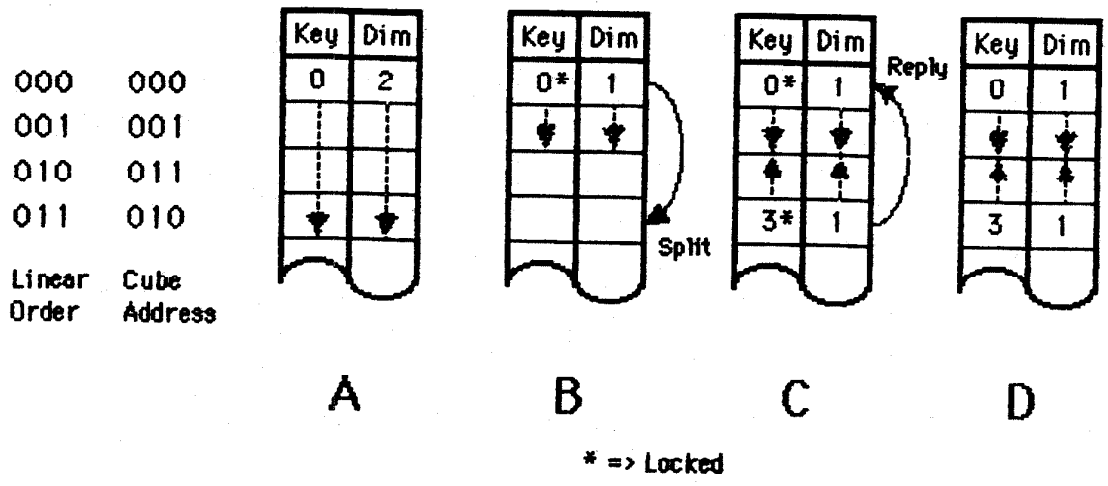

Figure 6: Insert Example

required. These readlocks, however, make the analysis of the operation simpler.

Example 4.1 Figure 6 shows the steps required to insert the key 3 into the cube of Figure 5. The search part of the insert proceeds as in Example 3.2 However, instead of terminating with a not found message, the key, 3, is inserted as follows:

1. Since the search must terminate in the UP direction, node $G(4)$ sends the search back to node $G(3)$. The state of the cube at this point is shown in Figure 6A.

2. As shown in Figure 6B, $G(0)$, the corner node of the $0XX$ subcube to which $G(3)$ belongs, decrements its dimension (from two to one), effectively detaching the $01X$ subcube, and sends a split message to its neighbor in dimension 1, node $G(3)$. $G(3)$ becomes the corner node of the newly formed subcube.

3. The split message inserts the key, 3, into node $G(3)$ and sets its dimension to 1 as shown in Figure 6C.

4. Finally, both nodes are unlocked as shown in Figure 6D.

Theorem 3 An insert operation in a stationary cube containing N nodes requires $O(\log N)$ time.

Proof: The initial stages of the insert are identical to the search operation and thus require $O(\log n)$ time. The final stage of the insert is the split operation which takes constant time. ■

Theorem 4 An insert operation will not deadlock with other concurrent operations.

Proof: While the insert operation can readlock two nodes simultaneously the second node locked is part of the subcube which is locked by the first lock. This second lock operation does not increase the number of nodes which are locked. Rather it is executed so that the upper half subcube will remain locked after its dimension is set to make it an independent subcube. This second subcube is in effect created by the insert and thus cannot previously have been locked by another operation. This node cannot be created by another process during the final stage of the insert since its corner node is locked, and the only way to create a node is to split it from its corner node. Thus, an insert operation will never have to wait to gain access to the split node. ■

5 Delete

Messages:

The delete operation is initiated by sending a delete message to any node in the cube. The delete operation searches for a node containing the delete key. If found, the operation marks this node as deleted, and replies to the requesting process. After the node is deleted, a continuously operating background process will attempt to merge the deleted node with an adjacent subcube to reclaim the deleted space. This background garbage collecting process runs on all corner nodes including deleted nodes.

DELETE	Requester	NodeAddress	Bounds	Key

The garbage collection process operates in two steps. The first step is to discover that an adjacent node is marked deleted. The second step is to recover the deleted node's space by merging it with its merge neighbor.

Definition 1 The *merge neighbor* of a node, $N[a]$, with address, a, is the node $N[m(a)]$ with address, $m(a) = a \oplus 2^{N[a].dim}$. If the subcubes cornered by nodes $N[a]$ and $N[m(a)]$ are of the same dimension, they can be merged to form a subcube of greater dimension. Further, node $N[m(a)]$ is the only node with which node $N[a]$ can be merged.

The discovery of an adjacent deleted node is accomplished by means of the `MERGEREQ` message. Deleted nodes periodically send `MERGEREQ` messages to their merge neighbors.

MERGEREQ	Node Address	Node Flag	Node Dimension

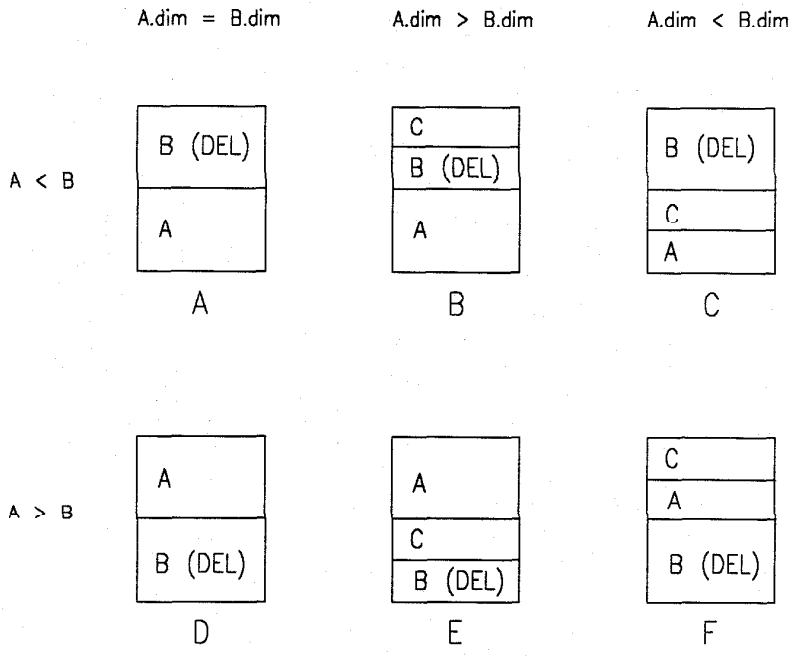

Figure 7: Merge Dimension Cases

Once a deleted node, B, is discovered its dimension is compared to the dimension of its merge neighbor, A. There are three possible cases as shown in Figure 7. If the two nodes are of the same dimension (Figure 7 (a,d)) they are merged. The garbage collector locks node A and sends a mergeup message or a mergedown message to B if A is below or above B respectively. These messages have the effect of extending the subcube cornered by node A to include the subcube cornered by node B. Since a merge operation must lock both nodes A and B a priority mechanism is used to prevent deadlock. If a mergedown message arrives at a node which is locked, it terminates unsuccessfully. A mergeup message will wait until the node is unlocked.

MERGEUP	Node Address

MERGEDOWN	Node Address	Node State

When the two adjacent nodes A and B have different dimensions a simple merge is not possible. There are two cases:

A.dim > *B.dim* : This case is not possible since B will not send a MERGEREQ message to A. As shown in Figure 7 (b,e), B is the merge neighbor of A but the converse is not

true. There is a third node C which is the merge neighbor of B. B sends its MERGEREQ message to node C, and it is up to node C to collect node B.

A.dim < *B.dim* : in this case we copy the contents of the linear address neighbor of node B to node B and mark this neighbor node deleted. If node A is above node B, this neighbor node will be node A itself (Figure 7 (f)). If node B is above node A the neighbor will be the node immediately below B (node C in Figure 7 (c)). In performing the copy we reduce the dimension of the deleted subcube and make it possible for the neighbor node A (C) to subsequently merge with its merge neighbor C (A) in Figure 7 (f (c)). The MOVE and COPY messages are used to move the contents of node A (C) to node B.

MOVE	Source Node Address	Destination Node Address

COPY	Node Address	Key	Record	Flag

Algorithm:

The delete algorithm is identical to the search until the key is found. Then the node is marked deleted. This has the result of routing all messages except a MERGE message addressed to this node to its merge neighbor.

```
/*
 *      Code to replace reply in search if key is found
 */
        N.deleted = TRUE ;                      /* mark deleted */
```

Eventually, the garbage collector running on a node, A, adjacent to the deleted node, B, will discover, by means of a MERGEREQ message, that B is deleted.

```
/*-------------------------------------------------------------*/
/*
 *      Garbage collector - This process is invoked by a
 *          MERGEREQ message from a deleted node.  It
 *          attempts to collect the deleted space by
 *          merging the deleted subcube.
 */
/*-------------------------------------------------------------*/
GarbageCollect(neighborAddress, neighborFlag, neighborDim)
{
  if(neighborFlag == deleted) {
    /* check if node A is lower half subcube */
    if((NodeAddress < neighborAddress) {
```

```
      /* if dimensions are the same we merge */
      if(N.dim == neighborDim) {
        LOCK ;
        if (send(neighborAddress,MERGEUP,WAIT) = TRUE) N.dim++ ;
        UNLOCK ;
      }
      /* if deleted subcube is larger we copy */
      else if (N.dim < neighborDim) {
        /* calculate address of node adjacent to deleted node */
        adjacentAddress =
        neighborAddress XOR 1<<neighborDim XOR 1<<(neighborDim -1) ;
        send(adjacentAddress,MOVE(neighborAddress)NOWAIT) ;
      }
    }
    /* otherwise node A is upper half subcube */
    else {
      /* if dimensions are the same we merge */
      if(N.dim == neighborDim) {
        READLOCK ;
        if (send(neighborAddress,
            MERGEDOWN(N.key,N.record,N.dim,N.flag),WAIT) == TRUE)
          N.flag = SLAVE ;
        READUNLOCK ;
      }
      /* if deleted subcube is larger we copy */
      else if (N.dim < neighborDim) {
        /* calculate address of node adjacent to deleted node */
        adjacentAddress =
        neighborAddress XOR 1<<neighborDim XOR 1<<(neighborDim -1) ;
        send(adjacentAddress,MOVE(neighborAddress)NOWAIT) ;
      }
    }
  }
}
/*-------------------------------------------------------------*/
```

The merge operation combines the subcube with the present node, A, at the corner with its adjacent subcube cornered by B. If the current node is the corner of the upper half subcube, the state of the current node is copied into the available lower half subcube with the mergedown message. If this message is successful, the current node flag is set to SLAVE to indicate that it is no longer a corner node. Since the nodes are inconsistent while the copying takes place, this operation is readlocked.

If the current subcube is below its adjacent subcube, then the current node is the corner of the combined subcube. In this case a mergeup message is sent to the adjacent subcube to set its flag to SLAVE. If this message completes successfully, the dimension of the current subcube is incremented to extend its domain over the merged subcube.

```
/*-------------------------------------------------------------*/
/*
 *      MergeUp(NodeAddress) - Waits for write access to node
 *      and then sets node to be a slave of requester.
 */
/*-------------------------------------------------------------*/
MergeUp(NodeAddress)
{
  LOCK ;
  N.flag = SLAVE ;
  send(requester,REPLY(TRUE),NOWAIT) ;
  UNLOCK ;
}

/*-------------------------------------------------------------*/
/*
 *      MergeDown(NodeAddress,key,record,flag) - Terminates
 *      immediately if write access cannot be acquired.  If
 *      access can be aquired, MergeDown READLOCKs the node
 *      and replaces its contents from the message.
 */
/*-------------------------------------------------------------*/
MergeDown(NodeAddress,key,record,flag)
{
  P ; /* semaphore to make this a test and optional set of the lock */
  if (N is LOCKED) {
    V ;
    send(requester,REPLY(FALSE),NOWAIT) ;
  }
  else { /* copy state */
    READLOCK
    V ;
    N.key = key ;
    N.record = record ;
    N.dim = N.dim + 1 ;
    N.flag = flag ;
    send(requester.REPLY(TRUE).NOWAIT) ;
    READUNLOCK ;
  }
}
/*-------------------------------------------------------------*/
```

The messages MOVE and COPY are used to move the contents of one node to another. When the MOVE message is received by a node, that node attempts to copy itself to destination of the move by sending a COPY message to the destination. If the COPY succeeds, it replies to the MOVE which then marks the source node deleted.

```
/*-------------------------------------------------------------*/
/*
 *      Move(addr) - Attempts to copy receiver to address.
 */
/*-------------------------------------------------------------*/
Move(addr)
{
  READLOCK ;
  if (send(addr,COPY(N.key,N.record,N.flag),WAIT)
    N.flag = DELETED ;
  READUNLOCK ;
}

/*-------------------------------------------------------------*/
/*
 *      Copy(key,record,flag) - Terminates immediately if
 *      destination is not deleted or free, or if the destination
 *      is locked.  Otherwise the contents of the destination
 *      are replaced by the contents of the message.
 */
/*-------------------------------------------------------------*/
Copy(key,record,flag)
{
  if(((N.flag == DELETED)||(N.flag == FREE))&&(NOT locked)) {
    READLOCK ;
    N.key = Key ;
    N.record = Record ;
    N.flag = flag ;
    READUNLOCK ;
    send(requester,REPLY(TRUE),NOWAIT) ;
  }
  else {
    send(requester,REPLY(FALSE),NOWAIT) ;
  }
}
/*-------------------------------------------------------------*/
```

Example 5.1 This example illustrates the simplest case of garbage collection where two nodes are the same size and all that is required is a merge. Figure 8A shows the state of a 2-cube where the key stored in $G(3)$ has just been deleted. To initiate collection, $G(3)$ sends a MERGEREQ message to its merge neighbor $G(0)$. The garbage collector then locks $G(0)$ and sends a mergeup message to $G(3)$ as shown in Figure 8B. This message locks $G(3)$. It will always succeed since mergeup messages have priority over mergedown messages. As shown in Figure 8C, the mergeup operation sets $G(3)$'s flag equal to SLAVE effectively attaching it to the $0X$ subcube. After the merge message replies, $G(0)$ increments its dimension to 2 to reflect the fact that the two subcubes, $1X$ and $0X$, have been merged to form a single subcube, XX. The final state of the subcube is shown in Figure 8D.

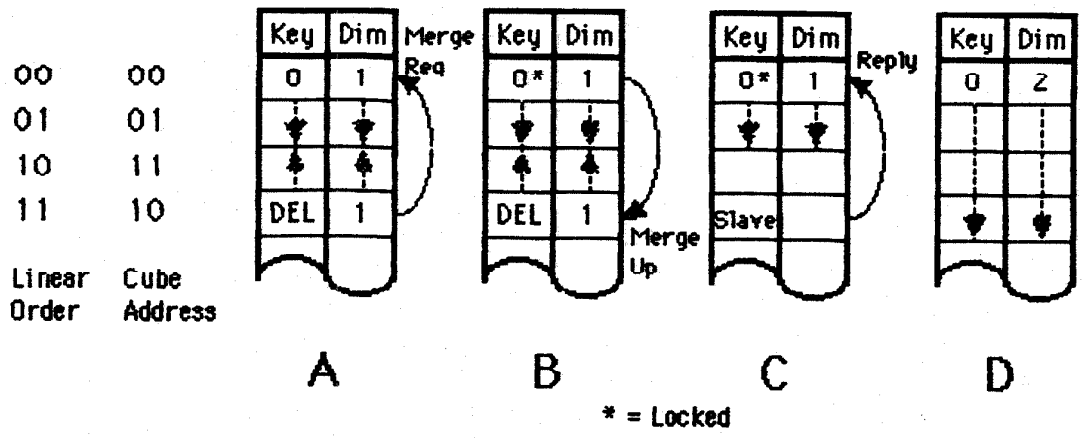

Figure 8: Merge Example: $A.dim = B.dim$

Example 5.2 This example, shown in Figure 9, illustrates the case where $A.dim < B.dim$ and A is above B. Node $G(3)/0$ (node $G(3)$ with dimension 0) receives a MERGEREQ message from node $G(0)/1$, as shown in Figure 9A. The linear address neighbor of $G(0)/1$ is calculated (by taking the exclusive-or of the node address with $2^{dim} + 2^{dim-1}$) to be $G(2)$. Node $G(3)$ sends a MOVE message to $G(2)$ as shown in Figure 9B. The MOVE locks node $G(2)$ and copies the key, record and flag from node $G(2)$ to node $G(0)$ by sending a COPY message as shown in Figure 9C. The COPY replies successfully to node $G(2)$, as shown in Figure 9D. Then node $G(2)$ is marked deleted. as illustrated in Figure 9E. Node $G(2)$ will now send a MERGEREQ to node $G(3)$ initiating the equal dimension garbage collection procedure.

Theorem 5 To delete a key from a cube with N nodes requires $O(\log N)$ time.

Proof: The search portion of the delete requires $O(\log N)$ time. Marking the node deleted and merging the node with its neighbor requires constant time. ∎

Theorem 6 The delete operation will not deadlock with other concurrent operations.

Proof: The delete operation only locks one node at a time. ∎

Theorem 7 The merge operations will not deadlock with other concurrent operations.

Proof: Although the merge operations lock two nodes simultaneously, this locking is ordered so that a node, A, will only wait for a node with an address greater than A to become unlocked. Thus, it is impossible to have a cycle of nodes waiting on each other's locks. ∎

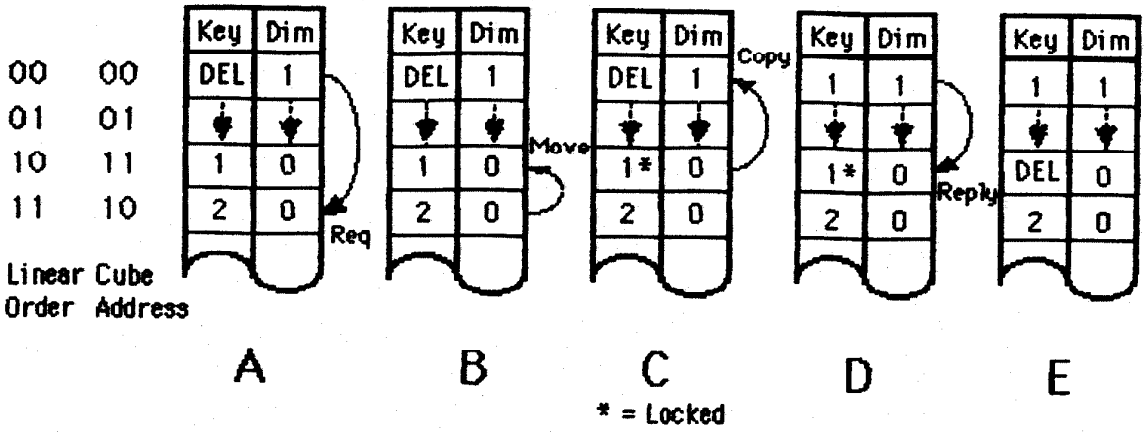

Figure 9: Merge Example: $A.dim < B.dim$, $A > B$

Before proving that concurrent search, insert, delete and merge operations will give the same result as running the operations sequentially in order of completion, we need to define some terms and prove one lemma about concurrent processes.

Definition 2 An operation *commits* when it has made a final decision to modify the state of a node in the cube and/or to reply with a particular result. Once an operation *commits* to modifying the state of a node, it must follow through and perform the modification. It cannot back out after committing.

Definition 3 The *commit condition* is the condition which must occur for an operation to commit.

Definition 4 An operation *completes* when it has finished modifying the state of a node. After an operation completes it cannot modify any additional state.

Definition 5 The *vulnerable period* of an operation is the period between the time it commits and the time it completes.

Definition 6 A *snapshot* of the cube is the state of all corner nodes of the cube with all processes stopped between critical regions. Since there is no concept of simultaneity between nodes of the cube, each process may be stopped at any point as long as causality and order of completion are preserved.

Definition 7 The *neighborhood* of an operation includes all nodes whose states are examined by the operation between the time it commits and the time it completes.

For example, a search operation commits and completes at the same time. A successful search commits to replying `FOUND` when it finds the requested data in the current node. An unsuccessful search commits to replying `NOTFOUND` when it receives a reply from a linear address neighbor confirming that the search key is not in the cube. An insert operation commits when the search portion of the insert receives the reply from the query message to an adjacent node. The commit condition is that the present node and the adjacent node bracket the insert key. The insert operation completes when the `Split` procedure unlocks its node. The node which is split constitutes the neighborhood of the insert operation. The commit condition for a delete operation is the key stored in the present node matching the delete key. When this condition is discovered the operation commits. A delete is completed when the delete flag of the node is set true. A merge commits when the mergeup or mergedown message is accepted. The commit condition is that the two nodes being merged are adjacent. Completion occurs when the merged node is unlocked.

Lemma 3 If an operation P's commit condition is valid throughout P's vulnerable period, and if P's neighborhood is not changed by another operation during this period, then any concurrent execution of P is consistent with a sequential execution of P ordered after all operations R which complete before P commits, ordered before all operations S which commit after P completes and ordered either before or after any operation Q which completes during P's vulnerable period.

Proof: P's commit condition and P's neighborhood constitute the state of the cube which is visible to P. If this state remains constant from the time P commits to the time P completes then P will act as if there were no concurrent operations, Q, during this period since it cannot see any changes caused by Q. It follows that P can be serialized with operations Q in any order. Since P's commit decision is valid after all operations R have completed, it will be valid if P is not started until after these operations have completed. Applying the same logic with S in place of P shows that operations S can be started after P completes without changing S's commit condition. ■

Theorem 8 Concurrent search, insert, delete and merge operations will give the same result as running the operations sequentially in order of completion.

Proof: The search, insert, delete and merge operations all meet the conditions in the hypothesis of Lemma 3:

Search completes at the time it commits and thus meets this condition. The commit condition for insert is that the present node, A, and the node directly above the present node, B, straddle the key to be inserted, K. This condition always holds at completion since: (1) a new node C<K cannot be inserted between A and B since this insert would have to be performed at A and A is locked, and (2) if B is deleted during this period, for any node D>B, D>K. The commit decision for delete is that the delete key is found. The node containing this key is locked so the condition still holds at completion. The commit condition for merge is that the adjacent node is marked deleted and the merge operation is able to lock the node. Since both of the nodes being merged are locked during the vulnerable period, this condition is still valid when the operation completes. For all these operations

the neighborhood is the present node which is locked and thus remains constant during the critical period. ■

6 Balance

The balancing process proceeds in three steps.

1. An imbalance between two adjacent subcubes, A and B, in the cube is recognized.
2. The subcube containing fewer data, say A, frees space on its border with B. Without loss of generality assume A is below B. To free space the node containing the highest datum in A, AH, splits itself freeing half its space.
3. The heavier subcube (containing more data), in this case B, moves its smallest datum to the space freed in step 2.

Imbalance is recognized by embedding a tree in the cube. As shown in Figure 10, for n=4, the tree is constructed by recursively dividing the cube into two subcubes. The node of each subcube closest in linear order to the other subcube is chosen as the corner node. This tree has one idiosyncrasy: messages to the outer child of a node must traverse two communication links while messages to the inner child of a node need to traverse only one link. Despite this shortcoming, however, the tree is ideal for balancing for two reasons. First, it evenly distributes the task of recognizing imbalance over all nodes of the cube except the zero node. Also, the root node of every cube is on the boundary of the cube across which a datum must be moved to balance the cube with an adjacent cube at the same level. Each root node participates in correcting an imbalance recognized by its parent.

The cube is balanced if for each internal node in the tree the number of keys stored in the subcubes represented by the two children of the node differ by less than $2:1$. Using the number of keys in a subcube as the balancing criteria rather than the maximum or minimum dimension of a node in the subcube has the advantage that local imbalances are averaged out when considering global balance.

Messages:

Leaf and internal nodes periodically transmit a size messages to their parent nodes. When the parent node receives the size message it updates its size and checks the sizes of its two subcubes for imbalance.

SIZE	Node Address	Root Node Address	Number of Keys

If the root node of a subcube detects imbalance between the two halves of its subcube, it initiates balancing by moving records between its two children. This data transfer takes place in two steps. First, a free message is transmitted to the boundary node of the subcube

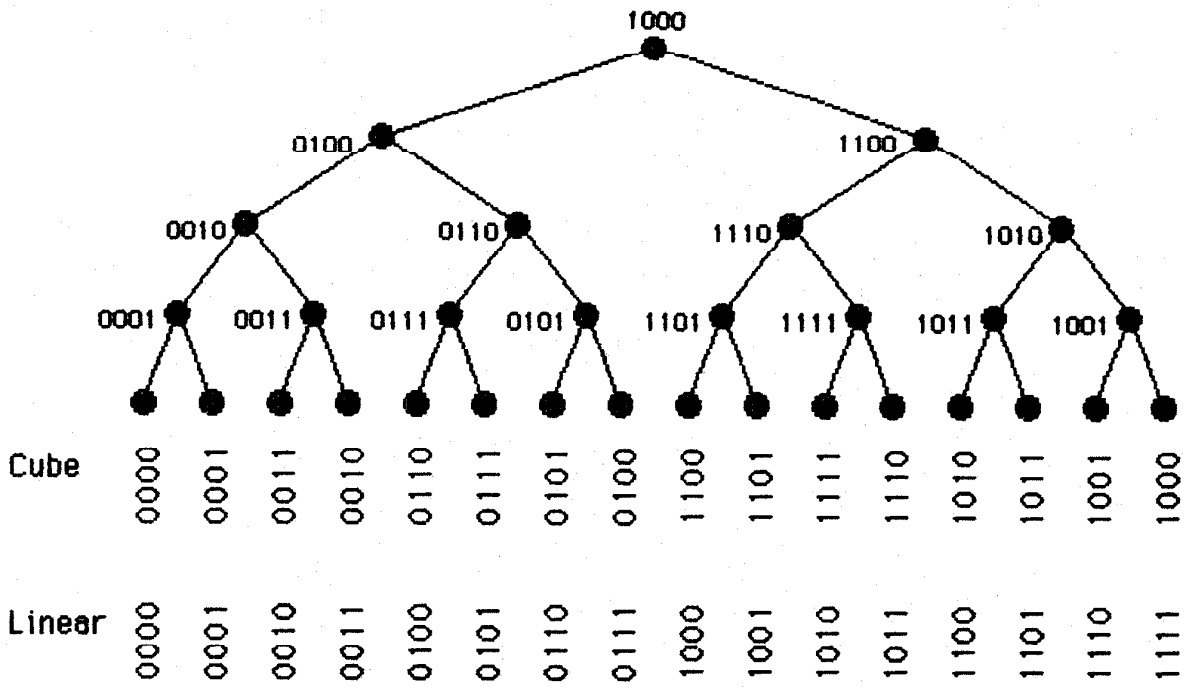

Figure 10: Balancing Tree, n = 4

containing fewer elements. This message causes the boundary node to split itself as in the insert operation, with the old key and record remaining in the node furthest from the subcube boundary. The boundary node of the subcube with the larger size is then sent a move message. This message locks the boundary node, copies its key and record to the freed node, and then marks the adjacent node deleted. The net effect is to move one datum from the larger subcube to the smaller subcube. While a node is marked free, it routes all its messages to the destination node. The root subcube repeats this operation until balance is restored to a 2:1 size ratio. It is important to note that because of the Gray code mapping most of these messages traverse only a single link in the cube. The message from the root to its outer child is the only message that must traverse two links.

FREE	Node Address	Adjacent Node Address

Algorithm:

The two balancing messages invoke three operations: update size, check balance, and free. The size message invokes the update size operation which in turn invokes the check balance operation. This procedure updates the size of the root subcube, checks for balance between its two subcubes and possibly initiates balancing by sending a free message. The free message invokes the free procedure which splits its destination node in half and sends a move message to the node in the other half subcube instructing it to copy itself to the freed node and then delete itself.

```
/*-------------------------------------------------------------*/
UpdateSize(RootNodeAddress, NodeAddress, Size)
{
  if(B(NodeAddress) < B(RootNodeAddress))   /* lower subcube */
    lower = Size ;
  else
    upper = Size ;
  rootSize = lower + upper ;
  CheckBalance(RootNodeAddress) ;
}
/*-------------------------------------------------------------*/
CheckBalance(RootNodeAddress)
{
  if(lower > 2*upper)
    send(lowerChild,FREE(upperChild),NOWAIT) ;
  else if(upper > 2*lower)
    send(upperChild,FREE(lowerChild),NOWAIT) ;
}
/*-------------------------------------------------------------*/
```

The free procedure shown below is similar to insert in that it must split the present node to generate a free block. There are two cases: If the subcube contains more than one element, the boundary node is a corner node. Since it is right on the boundary, it must copy its present state into the split subcube and then free itself. If the subcube contains only a single element, the boundary node is a slave to the root which recognized the imbalance. In this case the root simply sends a split message to free the boundary half of its subcube. As with insert, locking two nodes simultaneously is permissible during a split since the two nodes were the same node at the time of the first lock and it is impossible for another process to attempt to lock the split subcube after the original subcube is locked.

```
/*-------------------------------------------------------------*/
Free(NodeAddress, AdjacentNodeAddress)
{
  if(N.dim > 0) {
    if(boundary node is a corner node) {
      READLOCK :
      N.dim-- ;
      neighborAddress = NodeAddress XOR 1<<N.dim ;
      send(neighborAddress,SPLIT(N.dim,N.key,N.record),WAIT) ;
      N.flag = FREE ;
      READUNLOCK
      send(AdjacentNodeAddress, MOVE(NodeAddress), NOWAIT) ;
    }
    else {
      N.dim-- ;
      send(NodeAddress,SPLIT(N.dim,FREE),WAIT) ;
      send(AdjacentNodeAddress, MOVE(NodeAddress), NOWAIT) ;
```

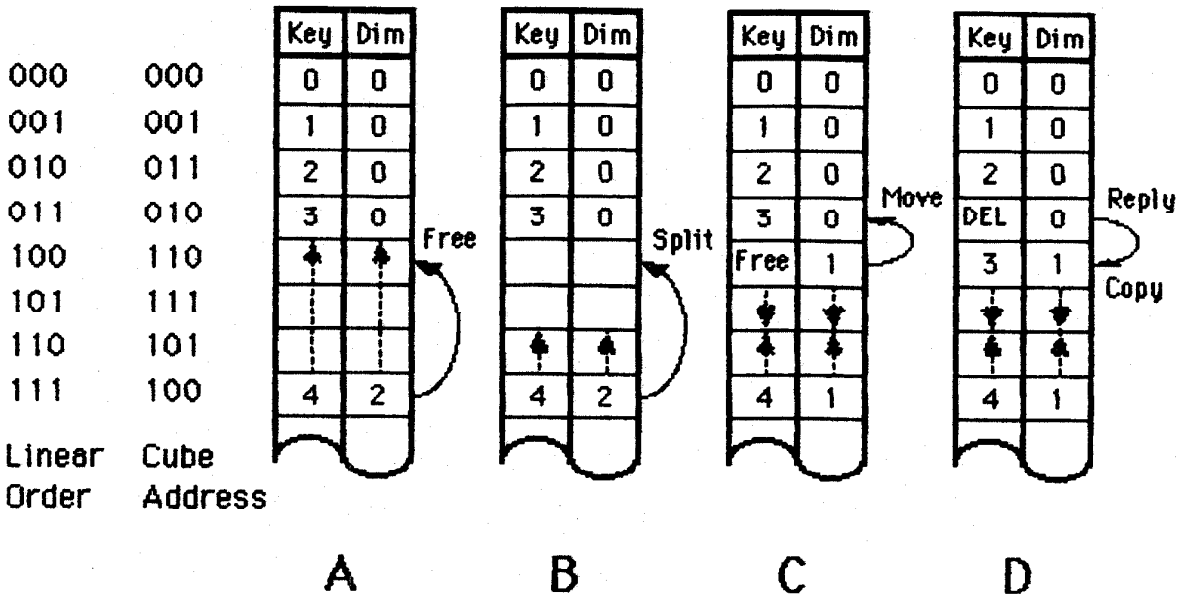

Figure 11: Balance Example

```
    }
  }
}
/*-------------------------------------------------------------*/
```

After a node is freed, the node which is to move to the freed subcube receives the move message. The move copies the boundary node's key and record to the freed node while preserving the freed node's dimension. After the copy completes the boundary node is marked deleted. Although two nodes are readlocked simultaneously, unlike the merge operation no priority resolution is required to prevent deadlock. Once a node is freed, there is only one node which can send a copy message to that node. Thus, as in the insert and free operations for purposes of locking, the freed node is part of the boundary node from the moment it unlocks after being tagged free.

Example 6.1 Figure 11 shows a balancing operation on a 3-cube. In Figure 11A, root node 100, $G(7)$, sees 1 record in the upper half of the cube and 4 records in the lower half of the cube. Recognizing this imbalance, $G(7)$ sends a free message to $G(4)$. As shown in Figure 11B, since $G(4)$ is a slave to $G(7)$, the free operation locks $G(7)$, decrements its dimension and sends a split message to $G(4)$. After the split message has marked $G(4)$ free, a move message is sent to $G(3)$ as shown in Figure 11C. After the move completes, $G(3)$ is marked deleted and the cube is balanced as shown in Figure 11D.

The balancing operations alter none of the arguments in the proofs of the theorems above. Thus, all of these theorems hold in a cube which is being dynamically balanced.

7 Extension to B-Cubes

A straightforward extension of the balanced cube is the B-cube. The B-cube is to a balanced cube what a B-tree is to a balanced tree. In the B-cube, rather than storing one record in each node, up to k records may be stored in each node. B-cube operations attempt to keep the number of records in each node between $\lceil \frac{k}{2} \rceil$ and k by splitting nodes when the number of records exceeds k and merging adjacent nodes when their combined number of records drops below $k+1$. Within a B-cube node, records are sorted and searched by conventional means. Between nodes the algorithms presented here for balanced cubes are applied with some modifications. For example in the search procedure a query message would reply with both upper and lower keys. The test for equality in this case would be $lower <= key <= upper$.

B-cubes have several advantages over balanced cubes:

- The overhead for maintaining the dimension and flag fields in each node is reduced. Rather than maintaining these fields for each record, their cost is spread out over up to k records. Locks in a B-cube can be either on a record basis or on a node basis. Write-locking at the node level and read locking at the record level seems to make the most sense.
- In a B-cube, the majority of inserts and deletes can be performed entirely within a single node without splitting or merging. Thus, the number of node interactions is reduced. Also, balancing is required less frequently since the number of operations which changes the node counts is reduced. Note, however, that when balancing is performed the amount of data to be moved has increased.
- It is expected that nodes will be swapped from a mass storage device. In the B-cube, the size of a node can be chosen to match a convenient transfer size for the storage device. In general this size is larger than a single record.

The single disadvantage of B-cubes is that they reduce the potential concurrency of the data structure. However, in most applications the number of records will greatly exceed the number of available processors and the concurrency of B-cubes will not be the limiting factor. In fact this reduction of concurrency is an advantage in the sense that it allows the granularity of the data structure to be smoothly varied over a large range.

8 Experimental Results

The balanced cube data structure has been implemented on a multiprocessor simulator and a number of experiments have been performed to verify the correctness of the algorithm and to measure their throughput. The balanced cube simulator is a 3000 line C program.[9] The code is divided fairly evenly into three parts:

- A binary n-cube simulator which provides the message passing environment of a concurrent computer.

- The balanced cube algorithms implemented in considerably more detail than presented in this paper. The interested reader should contact the author for a copy of this code.

- Instrumentation code to configure the cube simulator and to measure the performance of the balanced cube algorithms.

The decision to use a simulator instead of an actual concurrent computer for these experiments was a difficult one. The Caltech Cosmic Cube was available and was ideally suited to run the balanced cube algorithms. The simulator was chosen over the Cosmic Cube, however, because it offered greater flexibility and ease of instrumentation. The simulator can model the behavior of a wide range of concurrent computers. Computers of any size from one processor to 2^{16} processors can be simulated. For the experiments described below, the simulator was configured as a binary n-cube $1 \leq n \leq 13$. New communication topologies, such as a linear connected cube, can be easily added to the simulator. Also, it is easy to model different weightings of processing time to communication time on the simulator.

Two sets of experiments were run. The first set of experiments, described in detail in [10], was performed on an early version of the balanced cube which directly mapped the elements of the ordered set to the nodes of a binary n-cube. The current balanced cube algorithms, using a Gray code mapping, were used in the second set of experiments. After a few experiments were run to verify that the insert, delete and balance operations consume only a modest portion of the cube's resources, all remaining experiments were performed using only the search operation.

Throughput experiments were run to determine if the data structure can achieve the predicted $O(\frac{N}{\log N})$ throughput. These experiments were run using a load model that applied a maximum uniform load to the cube. The experiments were run for both the direct mapped cube and the current balanced cube.

The throughput results for the original direct mapped cube of [10] ,shown in Figure 12, fail to achieve the predicted throughput. The direct mapped cube only achieves a throughput of $O(\frac{N}{\log^2 N})$.

The degradation of $O(\log N)$ is due the non-uniformity of the hamming distance between linear order neighbors as expressed in equation (8). The function, d_{HA}, can be thought of as a barrier function. Shown in Figure 13, this function represents how many channels a message between linear address neighbors must traverse. Degradation occurs because the channels corresponding to the higher barriers must carry more traffic than the channels corresponding lower barriers. Hence these channels become congested.

The average barrier height is given by:

$$d_{HAL} = \frac{\sum_{i=1}^{n} i 2^{n-i}}{2^n} = \frac{2^{n+1}+2}{2^n} \approx 2. \tag{14}$$

The degradation is the ratio of maximum barrier height to average barrier height or $\approx \frac{n}{2}$. The experimental data of Figure 12 agrees exactly with this figure.

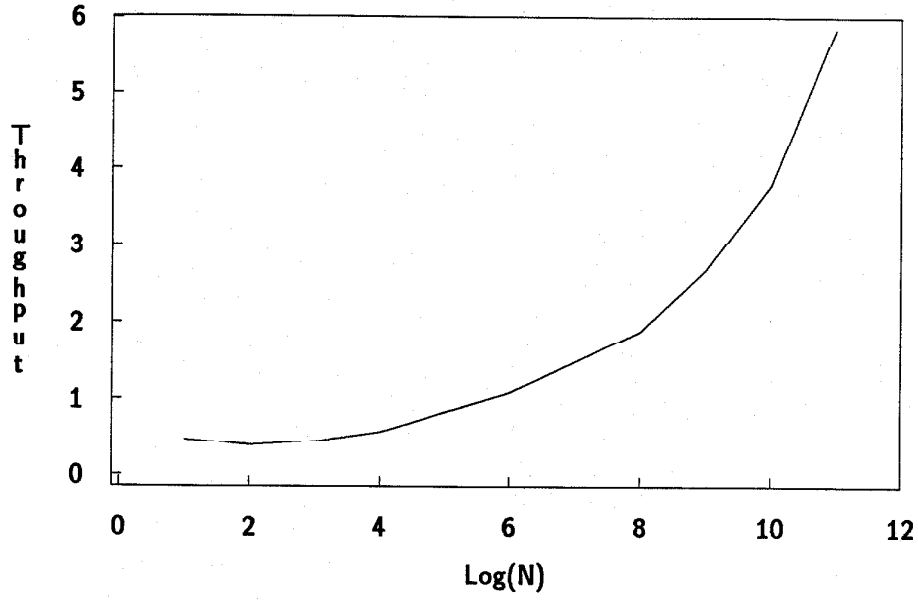

Figure 12: Throughput vs Cube Size for Direct Mapped Cube

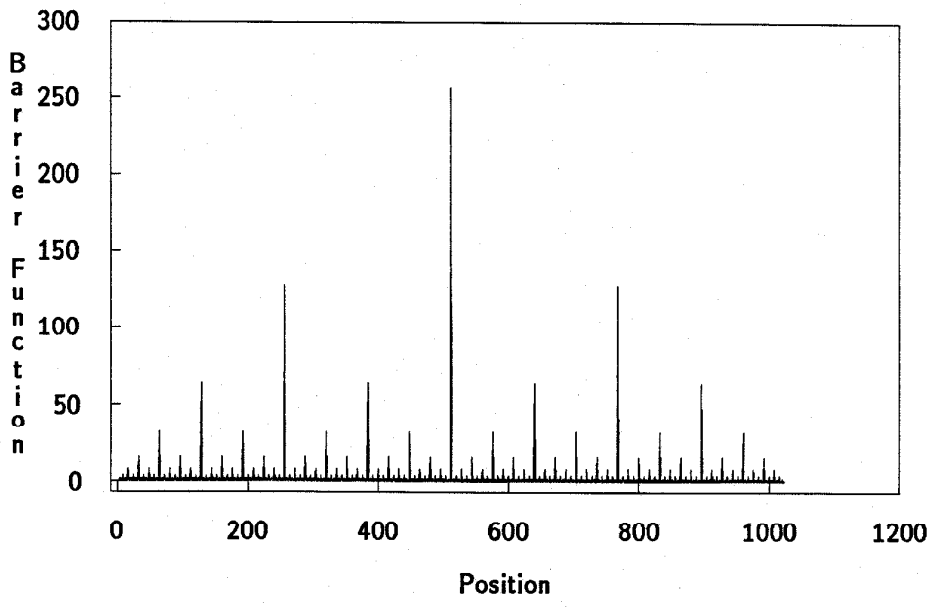

Figure 13: Barrier Function (n=10)

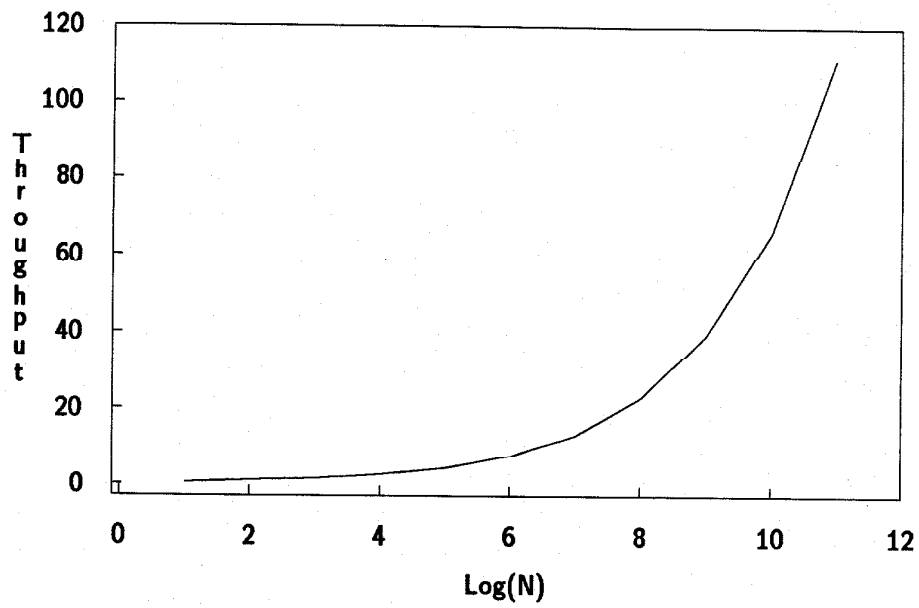

Figure 14: Throughput vs Cube Size for Balanced Cube

The Gray code mapping used in the current balanced cube eliminates this degradation as shown in Figure 14. The difference of $O(\log N)$ between Figures 12 and 14 illustrate the importance of developing data structures which match the topology of concurrent computers.

9 Conclusion

We have developed a new data structure for implementing ordered sets, the balanced cube. The balanced cube offers significantly improved concurrency over conventional data structures such as heaps, balanced trees and B-trees.

On sequential machines complexity is measured by instruction counts. Based on these conventional measures, the balanced cube performs as well as balanced trees or B-trees requiring $O(\log N)$ time to search, insert or delete a record in a structure of N records. For concurrent machines, however, communications costs are more important than instruction counts and the throughput of several operations executing in parallel is more important than the latency of a single operation. Based on this performance model, a balanced cube offers $O(\frac{N}{\log N})$ throughput as compared to $O(1)$ throughput for conventional data structures.

In any concurrent system consistency of interacting operations and deadlock avoidance are critical. The balanced cube is provably deadlock free. Each operation locks at most one non-deleted node at a time and unlocks this node before locking the next node. In the

case of the merge operation where there may be competition for access to deleted nodes a priority scheme is used to resolve any conflicts. In the balanced cube concurrently executing operations produce results which are consistent with a sequential execution of the same operations ordered by time of completion. This consistency is achieved by the judicious use of locking to make the completion of an operation appear instantaneous, and to assure that the neighborhood of an operation is not modified between the time it commits to modifying the state of the cube and the time it completes, performing the modification.

The balanced cube is a distributed ordered set object. It is an ordered set of data along with operations to manipulate those data distributed over the nodes or processors of a concurrent machine. Operations are initiated by messages to any node. Thus many operations may be initiated simultaneously. As concurrent machines become more common it is expected that other objects such as graphs, search trees, etc... will be implemented in a distributed manner. These objects, in turn, will motivate the architectures of the next generation of concurrent computers.

10 References

[1]Seitz, Charles L., "The Cosmic Cube," *CACM*, 28(1), January 1985, pp. 22-33.

[2]Aho, A.V., Hopcroft, J.E. and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974, pp. 87-92.

[3]Kung, H.T. and Lehman, P.L., "Concurrent Manipulation of Binary Search Trees," *ACM Transactions on Database Systems*, September 1980, pp. 354-382.

[4]Lehman, P.L and Yao, S.B., "Efficient Locking for Concurrent Operations on B-Trees," *ACM Transactions on Database Systems*, December 1981, pp. 650-670.

[5] Ellis, C.S., "Concurrent Search and Insertion in AVL Trees," *IEEE Transactions on Computers*, Vol. C-29, No. 9, September 1980, pp. 811-817.

[6] Ellis, C.S., "Concurrent Search and Insertion in 2-3 Trees," *Acta Informatica*, Vol 14, 1980, pp. 63-86.

[7]Sullivan, H. and Brashkow, T.R., "A Large Scale Homogeneous Machine," *Proc. 4th Annual Symposium on Computer Architecture*, pp 105-124, 1977.

[9] Kernighan, B.W. and Ritchie, D.,*The C Programming Language*, Prentice-Hall, 1978.

[10] Dally,W.J. and Seitz, C.L. *The Balanced Cube: A Concurrent Data Structure*, Caltech Technical Report 5174:TR:85, March 1985.