NISTIR 4637

Computer Implementation of a Discrete Set Algebra

Leonard Gallagher

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Computer Systems Laboratory
Database and Graphics Group
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4637

Computer Implementation of a Discrete Set Algebra

Leonard Gallagher

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards
and Technology
Computer Systems Laboratory
Database and Graphics Group
Gaithersburg, MD 20899

July 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
John W. Lyons, Director

Computer Implementation of a Discrete Set Algebra

Leonard Gallagher
Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

July 1991

Abstract

Large finite sets occur naturally in computer data management. Discrete elements such as numbers, pointers, icons, and object identifiers all have fixed-length bit sequence representations that may be viewed as base-two integers. The efficient storage and manipulation of large collections of such items is a long-standing problem in computer science. In particular, many data management algorithms may be specified in terms of set operations (e.g. union, intersection, crossproduct) on these collections. Often a large set is stored in reduced form for storage efficiency. A specific problem is then to find efficient algorithms for performing set operations over the reduced representations. This paper presents a binary tree storage mechanism for efficient representation of arbitrary sets of discrete elements taken from a fixed universe. It then develops high-level algorithms for mapping these sets to and from their tree representations and for executing set operations directly on the storage representation itself. This approach provides a method for direct computer implementation of mathematical models based on Boolean algebras defined over finite sets.

Key words: Algorithm, binary tree, Boolean algebra, data management, data structure, labeled tree, mathematical model, partial ordering, set operations, set theory.

References

1. Len Gallagher, "Computer implementation of an integer set algebra", Unpublished NBS technical report, January 1983.

2. ISO/IEC CD 11404, "Common language-independent data types (CLID)", Working Draft #5, document ISO/IEC JTC1/SC22/WG11 N233, 9 May 1991.

Table of Contents

Computer Implementation of a Discrete Set Algebra

by

Leonard Gallagher
NIST

1. Introduction

Reference [2] proposes a standardized collection of primitive and generator datatypes to be used for data sharing and application interoperability in a mixed-language programming environment. One of the proposed generator datatypes is a SET generator that creates a new datatype whose value space consists of all possible sets of elements from some specified base datatype with a finite number of elements. This paper is a modification and an enhancement of [1] that presents a theoretical basis for efficient representation, storage, and manipulation of data values defined by the SET datatype.

For small sets, the SET datatype can be implemented in a variety of different ways because efficiency is not a major consideration. For larger sets, such as those that occur naturally in data management and graphics applications, and in geographic information systems, more efficient storage structures and operational techniques are required. Discrete elements such as numbers, pointers, icons, and object identifiers all have fixed-length bit sequence representations that may be viewed as base-two integers. Thus the problem of implementing set algebras over such discrete bases reduces to the problem of implementing a set algebra over the universe of integers defined by fixed-length binary sequences.

Often a large set is stored in reduced form for storage efficiency. A specific problem is then to find efficient algorithms for performing set operations and evaluating predicates over the reduced representations. This paper presents a binary tree storage mechanism for representing sets of integers taken from a fixed universe. It then develops high-level algorithms for mapping these sets to and from their tree representations and for executing set operations directly on the storage representation itself. This approach provides a method for direct computer implementation of mathematical models based on Boolean algebras defined over finite sets.

Section 2 presents the operations and predicates of the SET datatype defined in [2]. Section 3 introduces a mathematical formalism for representing sets of integers as binary, labeled trees. Section 4 defines the data structures used to implement and navigate these trees. Sections 5 and 6 specify algorithms for traversing binary trees and for translating between sets and their labeled-tree representations. Sections 7 and 8 present algorithms for the set operations (i.e. union, intersection, and complement) and predicates (i.e. element and set containment) by operating directly on the tree representations without decompression or backtracking. Section 9 considers extensions to handle cross-products. An Appendix defines the mathematical terminology used in Section 3. All terms in bold face type are defined either in the text or in the Appendix.

2. The SET datatype

The SET datatype generator is specified in [2] as "SET OF <base>" where <base> is any discrete datatype already known to the implementation. SET creates a new datatype whose value space consists of values from the power set of the base type, with operations appropriate to a mathematical set algebra. The power set is the set of all subsets of the base type, including the empty set and the whole Universe of values from the base type. The SET datatype is subject to the following predicates and operations:

Setof(y: base): set of base

An operation that returns the set consisting of the single value y from the value space of <base>.

IsIn(x: base, y: set of base): boolean

A predicate that returns true if the value x is an element of the set y, and returns false otherwise.

Subset(x,y: set of base): boolean

A predicate that returns true if, for every value v of <base>, IsIn(v,x) implies IsIn(v,y), and returns false otherwise.

Equals(x,y: set of base): boolean

A predicate that returns true if Subset(x,y) and Subset(y,x), and returns false otherwise.

Complement(x: set of base): set of base

An operation that returns the set consisting of all values v from the value space of <base> such that IsIn(v,x) is false.

Union(x,y: set of base): set of base

An operation that returns the set consisting of all values v from the value space of <base> such that IsIn(v,x) or IsIn(v,y).

Intersection(x,y: set of base): set of base

An operation that returns the set consisting of all values v from the value space of <base> such that IsIn(v,x) and IsIn(v,y).

Empty(): set of base

A niladic operation that returns the empty set, i.e. the set that consists of no values from the value space of <base>.

Universe(): set of base

A niladic operation that returns the universal set, i.e. the set that consists of every value from the value space of <base>.

3. Mathematical formalism

Consider an integer set algebra $<P(\Omega); \cup, \cap, \neg>$ where, given a positive integer N, Ω is the set of all nonnegative integers from 0 to 2^N -1 and $P(\Omega)$ is the **power set** of Ω. The symbol $\cup$ is the union operator on $P(\Omega)$, $\cap$ the intersection operator, and $\neg$ the complement. We assume that $\cup$, $\cap$, and $\neg$ satisfy the axioms of a **Boolean algebra**. The purpose of this paper is to present a binary tree representation for subsets of Ω together with high-level algorithms for implementing the Boolean operations and forming and displaying sets.

For $x \in \Omega$, the binary sequence representation of x is $x_1, \ldots, x_N$ where $x_i \in \{0,1\}$ and

$$x = \sum_{i=0}^{N-1} 2^i \, x_{N-i}$$

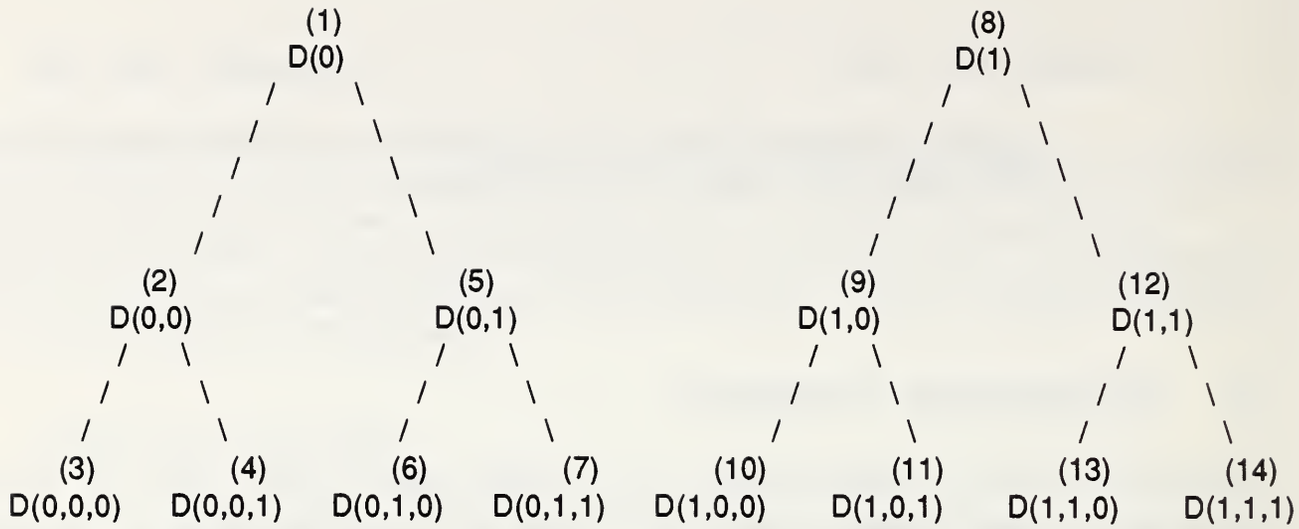
We define special subsets of Ω to be **dyadic intervals** as follows. A dyadic interval is denoted by $D(p_1,\ldots,p_m)$ where $p_i \in \{0,1\}$ and $m \leq N$; an integer x is an element of $D(p_1,\ldots,p_m)$ if $x_i = p_i$ for $1 \leq i \leq m$. Under the above description, if N=4, the set Ω is all integers from 0 to 15; the integer 9 is represented by the sequence 1,0,0,1; the dyadic intervals D(0,1) and D(1,0,1) represent the sets {4,5,6,7} and {10,11} respectively; and the dyadic interval D(1,1,0,1) represents the singleton set {13}.

The set T of all dyadic intervals is **partially ordered** by $\leq$ where $D(p_1,\ldots,p_m) \leq D(q_1,\ldots,q_n)$ if and only if $m \leq n$ and $p_i = q_i$ for i=1,...,m. It is easily seen that set inclusion of sub-intervals motivates the partial ordering and that under this partial ordering T is a **binary tree** of **height** N. T can also be **linearly ordered** lexicographically in a way that extends the partial ordering. That is $D(p_1,\ldots,p_m) < D(q_1,\ldots,q_n)$ if :

1. $m < n$ and $p_i = q_i$ for i=1,...,m , or if

2. for some r satisfying $0 \leq r < \min(m,n)$, $p_i = q_i$ for i=1,...,r, $p_{r+1} = 0$, and $q_{r+1}=1$.

In Figure 1 we give a graphical representation for the case when N = 3 and the set T has 14 elements. Each node is numbered to indicate linear rank, whereas diagonal lines from top to bottom represent the partial ordering.



Linear and Partial Orderings of T
Figure 1

Given a subset $A \subseteq \Omega$, we intend to represent A as a **labeled subtree** of T. We begin by defining a subtree T(A) of T recursively as follows:

1) $D(0) \in T(A)$ and $D(1) \in T(A)$

2) If $D(p_1,..., p_m) \in T(A)$ and $D(p_1,...,p_m) \cap A \neq \varnothing$ and $D(p_1,...,p_m) \cap \neg A \neq \varnothing$, then $D(p_1,..., p_m,0) \in T(A)$ and $D(p_1,..., p_m,1) \in T(A)$

T(A) inherits both the linear ordering and partial ordering of T, and under the partial ordering is a subtree of T. The **height** of T(A) is 1 if $A = \varnothing$ or if $A = \Omega$. The **height** of T(A) is N if A is a singleton set or if A contains any element that is "separated" from other elements of Ω.

Denote by $L_{T(A)}$ the mapping from the set of nodes, T(A), to the three element set {0, @,1} defined as follows:

1) $L_{T(A)}(D(p_1,..., p_m)) = 0$ if $D(p_1,..., p_m) \subseteq \neg A$

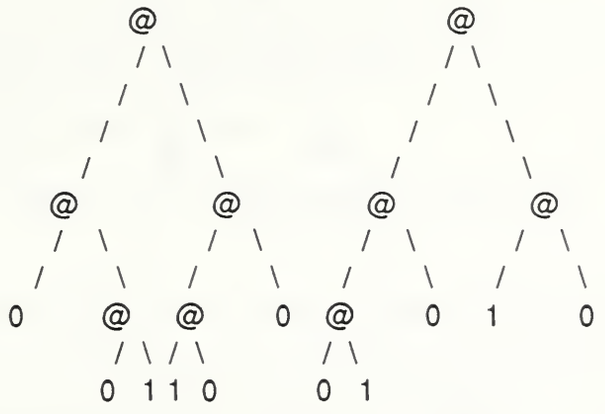
2) $L_{T(A)}(D(p_1,..., p_m)) = 1$ if $D(p_1,..., p_m) \subseteq A$

3) $L_{T(A)}(D(p_1,..., p_m)) = @$ otherwise

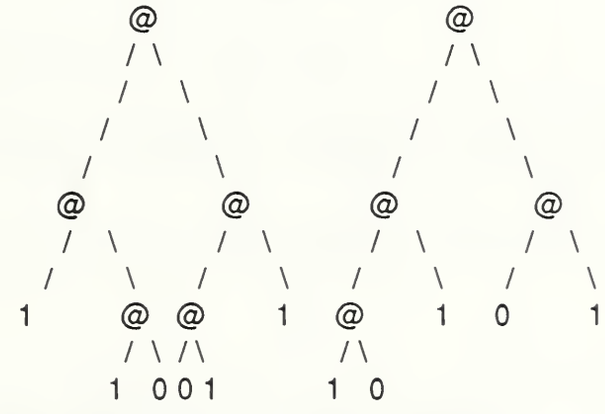
$L_{T(A)}$ is a set of ordered pairs, where the first element from each pair is a node of T(A) and the second element is a label from {0,@,1}. The partial ordering of T(A) extends naturally to $L_{T(A)}$ (see Appendix) making $L_{T(A)}$ a {0,@,1}-labeled tree. It is easy to see that there is a one-to-one correspondence between subsets of Ω and labeled trees constructed in this manner.

As an example of set representation by labeled trees, let N = 4 and consider the set A = {3,4,9,12,13} and its complement $\neg A$ = {0,1,2,5,6,7,8,10,11,14,15}. Both $L_{T(A)}$ and

$L_{T(\neg A)}$ are {0,@,1}-labeled trees of height 4. The graphical representations of A and $\neg A$ are given in Figures 2a and 2b below. The underlying dyadic intervals, $D(p_1,..., p_m)$, are omitted from the graphical representation since they can be reconstructed from the position of any node in the tree.



Labeled tree representation of A = {3,4,9,12,13}
Figure 2a



Labeled tree representation of $\neg$ A = {0,1,2,5,6,7,8,10,11,14,15}
Figure 2b

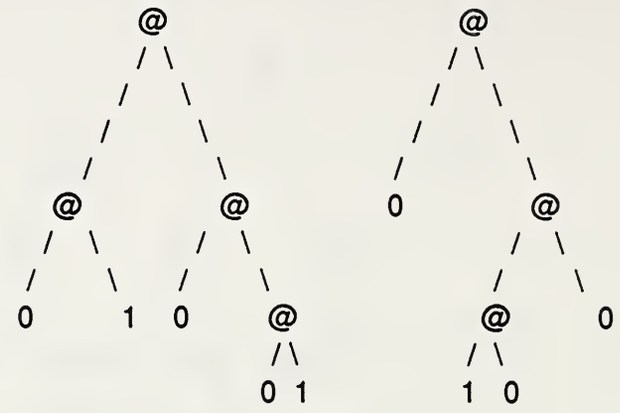
From these two tree representations we see that a set and its complement are represented by the same subtree. Only the labelings differ on the 0 and 1-labeled nodes. This means that the set complement operation will have a very straight-forward implementation algorithm.

The linear ordering on T(A) also extends naturally to $L_{T(A)}$ so a set A can be represented as a finite list of its labels. For example, the set A graphically represented in Figure 2a above could be structurally represented by the following list of labels:

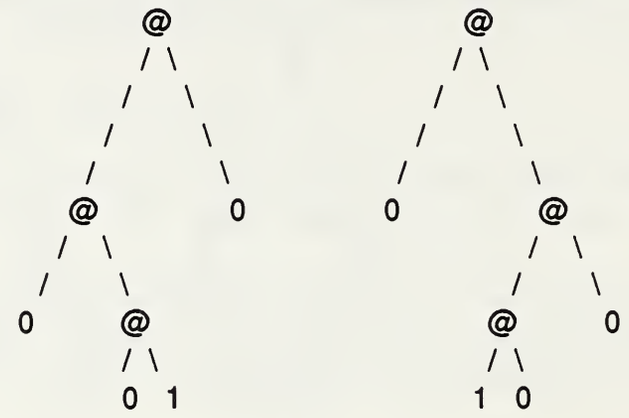
A $\Leftrightarrow$ @,@,0,@,0,1,@,@,1,0,0,@,@,@,0,1,0,@,1,0

We present an algorithm (Algorithm 4) to show that this procedure is reversible, thus proving that every subset of Ω is represented uniquely as a finite list of the labels {0,@,1} in this manner.

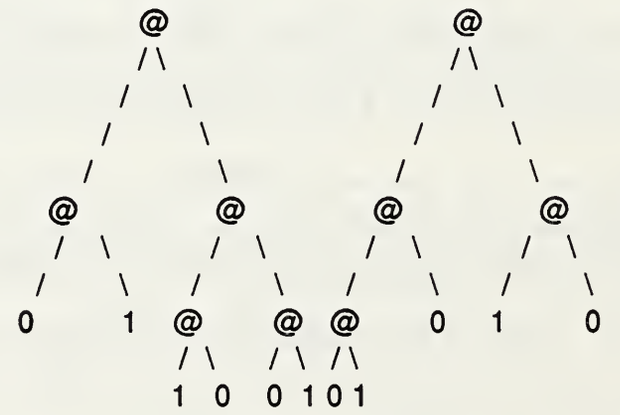
As an additional example, consider the set B = {2,3,7,12}. The graphical representations of B, $A \cap B$, and $A \cup B$ are given in Figures 3a, 3b, and 3c below.



Labeled tree representation of B = {2,3,7,12}
Figure 3a



Labeled tree representation of $A \cap B$ = {3,12}
Figure 3b



Labeled tree representation of $A \cup B$ = {2,3,4,7,9,12,13}
Figure 3c

From the above graphical representations it is easily seen that the structural representations of the sets B, $A \cap B$, and $A \cup B$ as lists of their labels are as follows:

$B \Leftrightarrow$ @,@,0,1,@,0,@,0,1,@,0,@,@,1,0,0

$A \cap B \Leftrightarrow$ @,@,0,@,0,1,0,@,0,@,@,1,0,0

$A \cup B \Leftrightarrow$ @,@,0,1,@,@,1,0,@,0,1,@,@,@,0,1,0,@,1,0

It is often useful to know the relationship between the cardinality of a given set and the length of its structural representation as a list of labels. From the above we see that the empty set and the universal set each have have a list representation of length two, i.e. $\varnothing \Leftrightarrow 0,0$ and $\Omega \Leftrightarrow 1,1$ respectively. A singleton set representation is of length 2N. In general, the length of the list of labels necessary to represent a set A, denoted by LENGTH(A), depends on four factors:

1) The size of the universe (i.e. the value of N)

2) The cardinality of the set (i.e. CARD(A))

3) The range of the set -- where range depends on the maximum and minimum values in the set and is defined as: RANGE(A) = MAX(A) - MIN(A) + 1

4) The distribution of A over its range interval -- where the range interval is defined as the set $\{x : MIN(A) \leq x \leq MAX(A)\}$.

The worst case occurs when A is uniformly distributed over its range interval and the set elements are "separated" as much as possible. For example, the set of even integers or the set of odd integers over any range would represent the worst case scenario. Using the worst case assumption, it is not too difficult to show that if m is the smallest integer satisfying

$$m \geq \log_2(RANGE(A)/CARD(A)$$

then

$$LENGTH(A) < 4N + 2(m+1)CARD(A).$$

In other cases, for example when all points in the set are contiguous, the length of the label string representation will be substantially less.

4. Data structures

The purpose of this section is to provide data structures for representing the integers, dyadic intervals, labels, and integer sets of the preceding section. We assume the existence of datatypes BIT and INTEGER, and define other datatypes together with the operations necessary to incorporate the algorithms of Sections 5 through 7. Whenever possible, data structure definitions use the terminology of [2].

Binary base integers

Integers are represented both by datatype INTEGER and by datatype BINARY. For a given N, datatype BINARY is defined to be ARRAY [1.. N] OF BIT. The assignment operation is used both to map integer values into binary values and to map components of binary values into bit values. That is, if X is a binary variable and x is an integer with binary representation $x_1,...,x_N$ as defined in Section 3, then X ⇐ x populates the array so that $X_i = x_i$ for $1 \le i \le N$. Also, Z ⇐ X(i) assigns the i-th bit value of X to the BIT variable Z.

Dyadic intervals

Dyadic intervals are represented by datatype DYADIC, where DYADIC is defined to be STACK OF BIT : SIZE(0,N). The dyadic interval $D(p_1,...,p_m)$ corresponds to the stack $p_1,...,p_m$ with p_m as the top of the stack. The operation "Height" maps dyadic variables with value $p_1,...,p_m$ into the integer m. We observe that Height(D) is always ≤ N and that Height(D) = N if and only if D is a dyadic interval representing a singleton set. The null stack, represented by ∅, is allowed and has height zero.

Operations on dyadic variables include Assignment (⇐), POP, PUSH, and TOP. The assignment operation assigns the stack value of one dyadic variable to another. The POP operation maps dyadic variables with value $p_1,...,p_m$ into the same variable with new value $p_1,...,p_{m-1}$ when $m > 1$, and with value equal to the null stack when $m = 1$. POP is not defined for the null stack. The POP operator also returns the top stack bit value and may be used in combination with assignment, for example Z ⇐ POP(D), to assign the top of D to the BIT variable Z. The PUSH operation maps a BIT variable Z and a dyadic variable D having value $p_1,...,p_m$ into the same variable D but with new value $p_1,...,p_{m+1}$ satisfying p_{m+1} = Z. We define a third operation, TOP, which returns the bit value of the top of the stack value for the dyadic variable, but leaves the stack value itself unchanged.

Predicates on dyadic variables include a check for equality (=) and a check for partial ordering (≤). Two dyadic variables with stack values $p_1,...,p_m$ and $q_1,...,q_n$ respectively are equal if and only if $m = n$ and $p_i = q_i$ for $i=1,...,n$. They satisfy the partial ordering defined in Section 3 if and only if $m \le n$ and $p_i = q_i$ for $i=1,...,m$.

Labels

LABEL is an enumeration consisting of just three elements: 0, @, and 1. Since order is not important, we define LABEL to be STATE(0,@,1). These elements are the three possible labels for labeled trees representing a set. In some algorithms, we need a fourth symbol to be the "End of File" marker that indicates termination of a list of labels. For these algorithms we use the

extension mechanism of [2] to define an extended label datatype as LABEL : EXTENDED(EOF). The only operations on LABEL variables are Assignment (⇐), and Equality (=).

List of Labels

The union and intersection set operation algorithms in Section 7 require a "list of labels" data structure to store and manipulate labels during set processing. Such lists will always be relatively short, never exceding length 2N, where N is defined in Section 3 and determines the cardinality, 2^N, of the set Universe. We specify a datatype LIST OF LABEL : SIZE(0,2N) to represent variable length lists of labels under the constraint that no list has length exceeding 2N labels. The null list, represented by ∅, is allowed and has size zero. Variables of this type are internal algorithm processing tools only and never require external representation.

Operations on these lists of labels include PUTFIRST, PUTLAST, GETFIRST, and GETLAST. The PUT operations append a new label to the "head" or "tail" of the list, respectively. The GET operations remove the head or tail elements from the list and result in a new list one unit shorter than the original list. The GET operations may be used together with the assignment operation to assign the head or tail elements of a label list to a label variable. For example, if L is the label-list $p_1,...,p_m$, then Z ⇐ GETFIRST(L) assigns p_1 to Z and sets L to $p_2,...,p_m$. A "Length" operation returns the length of any such list.

Set of integer

We represent integer sets by the datatype SET, where SET is defined to be LIST OF LABEL. A set variable has as its value a list of labels representing some integer set as constructed in Section 3. The only allowed operations on SET datatypes are the "Head" and "Append" operations defined for lists in [2]. A set must be in either "read" mode or "build" mode. Head is legal only when the set is in read mode and Append is legal only when the set is in build mode.

We define two functions mapping set variables into integers. CARD(B) returns the cardinality of the set represented by B and LENGTH(B) returns the length of the label string . In addition we define several predicates over set variables. These are discussed in Section 8.

Pointer to set of integer

We require a SET_POINTER datatype that is able to traverse the list of labels of a SET datatype from head to tail and maintain its position in the set without modifying the set itself. A set_pointer is initialized to the first position of the label-list representing a set whenever a set variable for that set is assigned to the set_pointer variable. Because the label-list representing a set is always traversed from head to tail with no backtracking and because the label-list is always built by appending just to its tail, the set_pointer is just a virtual implementation of the "Head" and "Append" operations defined for lists.

Operations on set_pointer variables include Assignment (⇐), GETLABEL, and PUTLABEL. The assignment operation assigns the current set and the current position in that set from one set_pointer variable to another. GETLABEL corresponds to the "Head" operation for lists. If $b_1,...,b_m$ is the value for a set variable B and if the set_pointer PB pointing to B has as its current position b_t, then GETLABEL acts on PB to return the label b_t. GETLABEL also sets the pointer to b_{t+1} when $t < m$ and to End-of-File (EOF) when $t = m$. GETLABEL is a valid

operation only when the set pointed to by the set_pointer variable is in "read" mode. PUTLABEL is an operation mapping a label variable Z and a set_pointer variable PB pointing to the set value $b_1,...,b_m$ into the same variable PB with new set value $b_1,...,b_{m+1}$ satisfying $b_{m+1} = Z$. PUTLABEL corresponds to the "Append" operation for lists. PUTLABEL is a valid operation only when the set pointed to by the set_pointer variable is in "build" mode.

Sometimes we have a need to exchange the values of two set_pointer variables. Rather than introduce a temporary set_pointer value, we accomplish the two way assignment with an "Exchange" operation. The result of EXCHANGE(PA, PB) is that the value of PB is assigned to PA and the value of PA is assigned to PB.

5. Tree traversal algorithms

We describe algorithms for two operations on elements of the tree T defined in Section 3. Algorithm 1 allows one to "skip over" a subtree of T. That is, if D is a variable of type DYADIC having as its value a stack of bits representing some node D_1 of T, then SKIP accepts the D_1 value as input and whenever possible returns to D a stack of bits representing the node D_2 of T satisfying:

1) $D_1 < D_2$ (i.e. D_2 is greater than D_1 in the linear order)

2) $\neg(D_1 \leq D_2)$ (i.e. D_1 and D_2 are not related in the partial order)

3) $D_1 < D_3 < D_2 \Rightarrow D_1 \leq D_3$ (i.e. D_2 is the next largest node not in the subtree of D_1)

If a node satisfying the above conditions does not exist, then SKIP returns the null stack ∅. Less precisely, we observe that SKIP returns the next largest node that is not in the subtree determined by the current node. For example, in Figure 1, SKIP acting on node 2 returns node 5, acting on node 10 returns node 11, acting on nodes 1, 5, or 7 returns node 8, and acting on nodes 8, 12, or 14 returns the null stack.

The ADVANCE algorithm, Algorithm 2, provides a successor function for traversing the linear ordering of T. It accepts some D value as input and, if possible, returns to D the stack representing the next largest dyadic interval (under the linear ordering for T). Otherwise, when the largest node has been reached, it returns the null stack.

Algorithm 1

SKIP accepts a node of T and returns the next largest node (under the linear ordering) that is not in the subtree determined by the given node. If no such node exists, the null stack is returned.

NODE	of type	DYADIC
Z	of type	BIT

```
begin
   while (Height(NODE) > 0)
      begin
         Z ⇐ POP(NODE) ;
         if  Z=0  then
            begin
               PUSH(1, NODE) ;
               RETURN
            end
      end
end
```

Restatement: POP the stack looking for the first 0 bit and change it to a 1 bit. If unsuccessful, return the null stack.

Algorithm 2

ADVANCE accepts a node of T and returns the next largest node (under the linear ordering). If the given node is the largest node of T, then the null stack is returned.

NODE of type DYADIC

```
begin
   if Height(NODE) = N
      then SKIP(NODE)      ** See Algorithm 1 **
      else PUSH(0, NODE)
end
```

Restatement: Return the left node from the next level of the tree. If already at highest level, get the right node. If already at the right node, SKIP to the next node at a new level.

6. Forming and displaying sets

We provide two basic algorithms for implementing transformations between integer sets and their label-list representations. Algorithm 3, named FORMSET, determines the label-list representation for a singleton set. It is an implementation of the SetOf operator defined in Section 2 when the base datatype is integer. For a fixed N it accepts an integer x satisfying $0 \leq x < 2^N$ and outputs the list of labels described in Section 3 for the set {x}. For example, when N = 4, FORMSET acts on 12 to produce the label-list 0,@,0,@,@,1,0,0 and acts on 5 to produce the label-list @,0,@,@,0,1,0,0. The resulting list of labels has length 2N and contains exactly N 0-labels, N -1 @-labels, and exactly one 1-label. The FORMSET algorithm taken together with the UNION algorithm (Algorithm 5) provides a method for building up set representations one element at a time.

The DISPLAY algorithm, Algorithm 4, is the major tool for transforming a label-list set representation into a sequence of integers. It traverses the tree T of dyadic intervals and determines whether or not an interval contains elements of an underlying set. Whenever an interval does not contain elements of the set, the algorithm employs SKIP to bypass all subintervals of that interval. If an interval contains elements of the set, the algorithm employs ADVANCE to systematically check subintervals. Then, whenever a singleton interval containing an element of the set is isolated, the EVALUATE function (4.1) is employed to return the single integer element of that interval. The integer is put into an output sequence and the algorithm continues its traversal of T. The advantage of this algorithm is that it reads through the bit

string representation only once and produces all elements (in ascending order) of the underlying set.

Algorithm 3

FORMSET accepts an integer x and returns the label-list representation for {x}. The label-list will always be of length 2N, and will contain exactly N 0-labels, N -1 @-labels, and one 1-label.

x	of type	INTEGER
X	of type	BINARY
NODE	of type	DYADIC
XSET	of type	SET
PX	of type	SET_POINTER

```
begin
   Put XSET into "build" mode ;
   PX ⇐ XSET ;
   X ⇐ x ;  NODE ⇐ ∅ ;
   PUSH(0, NODE) ;                          ** Form the initial dyadic interval **
   while NODE ≠ ∅                                ** Exit when node is null **
      if TOP(NODE) = X(Height(NODE)) then          ** Is x in NODE **
         begin
            if Height(NODE) < N then               **Not a leaf node**
               begin
                  PUTLABEL(@, PX);
                  PUSH(0, NODE)
               end
            else                                   **If leaf node**
               begin
                  PUTLABEL(1, PX);
                  while Length(XSET) < 2N          **Fill with 0's**
                     PUTLABEL(0, PX);
                  NODE ⇐ ∅                         **Set exit flag**
               end
         end
      else
         begin
            PUTLABEL(0, PX) ;
            SKIP(NODE)                             ** Algorithm 1 **
         end
end
```

Note: The FORMSET algorithm is easily extended to accept as input a variable of type DYADIC and to output the label-list of the set determined by that dyadic interval.

Algorithm 4

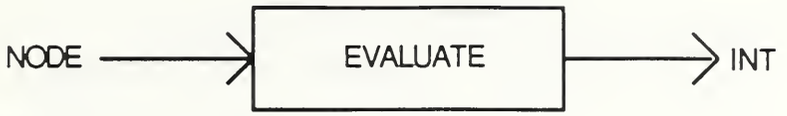
DISPLAY accepts a label-list representation for some set as input and returns an output sequence containing the integer elements of the set.

STRING	of type	SET
PS	of type	SET_POINTER
ELEMENTS	of type	LIST OF INTEGER
NODE	of type	DYADIC
MARK	of type	DYADIC
Z	of type	LABEL

```
begin
    Put STRING into "read" mode ;
    PS ⇐ STRING ;
    NODE ⇐ ∅ ;
    PUSH(0,NODE) ;                    ** Form the initial dyadic interval **
    while NODE ≠ ∅                    ** Exit when node is null **
        begin
            Z ⇐ GETLABEL(PS) ;
            if Z = @ then
                PUSH(0, NODE)                    **Get next interval**
            else if Z = 0 then                   **Not overlapping interval**
                SKIP(NODE)
            else                                 **Set contains interval**
                begin
                    MARK ⇐ NODE;
                    SKIP(MARK);
                    while NODE ≠ MARK
                        begin
                            while Height(NODE) < N          **Get next leaf**
                                PUSH(0, NODE);
                            Append(ELEMENTS, EVALUATE(NODE));**Function 4.1**
                            SKIP(NODE)                      **Algorithm 1 **
                        end
                end
        end
end
```

Function 4.1

EVALUATE accepts a dyadic variable of height N (i.e. representing a singleton integer set) as input, and returns the integer itself. The value of the input variable is left unchanged.



NODE, BINREP	of type	DYADIC
INT, J, K	of type	INTEGER

```
begin
    BINREP ⇐ NODE ;
    INT ⇐ 0 ;
    for J = 0 to N-1
        begin
            K ⇐ POP(BINREP) ;
            INT ⇐ INT + K * 2**J
        end
end
```

Restatement: Read BINREP from bottom to top as the binary representation of the desired integer. Multiply by the appropriate power of two and sum the products.

7. Set operations

The basic operations of a set algebra are union, intersection, and complement. Other operations like set difference and symmetric difference are defined in terms of the basic three. To this end we present algorithms for union, intersection, and complement. For reasons of efficiency, however, it is sometimes helpful to have separate algorithms for each operation used often in any application. It is easily seen that a separate algorithm for set difference differs only slightly from the algorithm for intersection and that a separate algorithm for symmetric difference combines some aspects of union and intersection.

These algorithms are similar and all use the main idea of Algorithm 4. That is, the algorithm traverses the tree T of dyadic intervals checking whether or not the result of the set operation intersects the current interval. If the interval intersects both the result and its complement, then a @ is put on its label-list representation. If the interval is a subset of the result, then a 1 is put on its label-list representation . If the interval is disjoint from the result, then a 0 is put on its label-list representation. In each case the alternative is determined by checking the input labels representing the set or sets upon which the set operation operates. The main advantage of this approach is that each label-list is traversed only one time from beginning to end; at no time is it necessary to backtrack or re-read labels. This makes it possible to take advantage of parallel processing architectures when evaluating complex set expressions.

Algorithm 5

UNION accepts label-list representations for two sets and returns the label-list representation for their set union. The two input lists are left unchanged.

A, B, C	of type	SET
PA, PB, PC	of type	SET_POINTER
Z1, Z2, Z	of type	LABEL
NODE, MARK	of type	DYADIC
BUFFER	of type	LIST OF LABEL : SIZE(0,2N)

```
begin
    Put A and B into "read" mode ;
    Put C into "build" mode ;
    PA ⇐ A;  PB ⇐ B;  PC ⇐ C;
    NODE ⇐ ∅;  MARK ⇐ ∅ ; BUFFER ⇐ ∅
    PUSH(0,NODE);
    while NODE ≠ ∅                          ** Exit when node is null **
        begin
            Z1 ⇐ GETLABEL(PA);
            Z2 ⇐ GETLABEL(PB);
            if (Z1 = 0  and  Z2 = 0) then
                begin
                    while (BUFFER ≠ ∅)  PUTLABEL(GETFIRST(BUFFER), PC);
                    PUTLABEL(0, PC);
                    SKIP (NODE)
                end
            else if (Z1 = @  and  Z2 = @) then
                begin
                    APPEND(@, BUFFER);                  **Algorithm  5.1**
                    ADVANCE (NODE)
                end
            else if ( Z1 = 1  or  Z2 = 1)  then

                begin
                    if  Z1 ≠ 1
                        then begin  EXCHANGE(PA, PB);  Z ⇐ Z1 end
                        else  Z ⇐ Z2 ;
                    if  BUFFER = ∅
                        then  PUTLABEL(1, PC)
                        else  APPEND(1, BUFFER) ;       **Algorithm  5.1**
                    MARK ⇐ NODE;
                    SKIP  (MARK);
                    if  Z = @  then
```

```
                    begin
                        ADVANCE (NODE);
                        while NODE ≠ MARK
                            if GETLABEL(PB) = @
                                then ADVANCE (NODE)
                                else SKIP (NODE)
                    end
                else SKIP (NODE)
            end
        else if (Z1 = 0 or Z2 = 0) then
            begin
                if Z1 ≠ 0 then EXCHANGE(PA, PB) ;
                MARK ⇐ NODE;
                SKIP (MARK);
                APPEND (@, BUFFER);                    **Algorithm 5.1**
                ADVANCE (NODE) ;
                while NODE ≠ MARK
                    begin
                        Z ⇐ GETLABEL (PB);
                        if Z = 0 then
                            begin
                                while (BUFFER ≠ ∅)
                                        PUTLABEL(GETFIRST(BUFFER), PC);
                                PUTLABEL (0, PC);
                                SKIP (NODE)
                            end
                        else
                            begin
                                if (BUFFER = ∅ and Z=1)
                                    then PUTLABEL(1, PC)
                                    else APPEND(Z, BUFFER) ;    **Alg 5.1**
                                if Z = @
                                    then ADVANCE (NODE)
                                    else SKIP (NODE)
                            end
                    end
            end
        end ;
    while (BUFFER ≠ ∅) PUTLABEL(GETFIRST(BUFFER), PC)
end
```

For set difference and symmetric difference we can use the separate algorithms discussed above or we can apply the following algebraic identities:

Set Difference $A - B = A \cap \neg B$

or

Symmetric Difference $A \oplus B = (A - B) \cup (B - A)$

Algorithm 5.1

APPEND accepts a single label and a variable length list of labels and returns a modified variable length list. The list is modified so that any would be trailing sublist of the form @,1,1 is replaced by 1 and any would be trailing sublist of the form @,0,0 is replaced by 0.

Z	of type	LABEL
LIST	of type	LIST OF LABEL : SIZE(0,2N)

```
begin
   if (Z = @ or Length(LIST) ≤ 1) then
      PUTLAST (Z, LIST)
   else if Z = GETLAST (LIST) then      **check for match**
      begin
         GETLAST (LIST);                **remove @ label**
         APPEND (Z, LIST)               **recursive call**
      end
   else
      begin
         PUTLAST (@, LIST);             **replace @ label**
         PUTLAST (Z, LIST)
      end
end
```

Restatement: An @ label is simply concatenated to the end of the list; otherwise, if the label to be added is identical to the last element of the list, then the last two elements of the list are replaced by the label to be added, recursively.

Algorithm 6

INTERSECTION accepts label string representations for two sets and returns the label string representation for their set intersection. The two input lists are left unchanged.

A, B, C	of type	SET
PA, PB, PC	of type	SET_POINTER
Z1, Z2, Z	of type	LABEL
NODE, MARK	of type	DYADIC
BUFFER	of type	LIST OF LABEL : SIZE(0,2N)

```
begin
   Put A and B into "read" mode ;
   Put C into "build" mode ;
   PA ⇐ A; PB ⇐ B; PC ⇐ C;
   NODE ⇐ ∅; MARK ⇐ ∅ ; BUFFER ⇐ ∅
   PUSH(0,NODE);
   while NODE ≠ ∅                          ** Exit when node is null **
      begin
         Z1 ⇐ GETLABEL(PA);
         Z2 ⇐ GETLABEL(PB);
         if (Z1 = 1 and Z2 = 1) then
            begin
               while (BUFFER ≠ ∅) PUTLABEL(GETFIRST(BUFFER), PC);
               PUTLABEL(1, PC);
               SKIP (NODE)
            end
         else if (Z1 = @ and Z2 = @) then
            begin
               APPEND(@, BUFFER);                  **Algorithm 5.1**
               ADVANCE (NODE)
            end
         else if ( Z1 = 0 or Z2 = 0) then
            begin
               if Z1 ≠ 0
                  then begin EXCHANGE(PA, PB); Z ⇐ Z1 end
                  else Z ⇐ Z2 ;
               if BUFFER = ∅
                  then PUTLABEL(0, PC)
                  else APPEND(0, BUFFER) ;         **Algorithm 5.1**
               MARK ⇐ NODE;
               SKIP (MARK);
               if Z = @ then
```

```
                    begin
                        ADVANCE (NODE);
                        while NODE ≠ MARK
                            if GETLABEL(PB) = @
                                then ADVANCE (NODE)
                                else SKIP (NODE)
                    end
                else SKIP (NODE)
            end
        else if (Z1 = 1 or Z2 = 1) then
            begin
                if Z1 ≠ 1 then EXCHANGE(PA, PB) ;
                MARK ⇐ NODE;
                SKIP (MARK);
                APPEND (@, BUFFER);                    **Algorithm 5.1**
                ADVANCE (NODE) ;
                while (NODE ≠ MARK)
                    begin
                        Z ⇐ GETLABEL (PB);
                        if Z = 1 then
                            begin
                                while (BUFFER ≠ ∅)
                                        PUTLABEL(GETFIRST(BUFFER), PC);
                                PUTLABEL (1, PC);
                                SKIP (NODE)
                            end
                        else
                            begin
                                if (BUFFER = ∅ and Z=0)
                                    then PUTLABEL(0, PC)
                                    else APPEND(Z, BUFFER) ;    **Alg 5.1**
                                if Z = @
                                    then ADVANCE (NODE)
                                    else SKIP (NODE)
                            end
                    end
            end
    end ;
while (BUFFER ≠ ∅) PUTLABEL(GETFIRST(BUFFER), PC)
end
```

Algorithm 7

COMPLEMENT accepts a label string representation for a set and returns the label string representation for its complement. The input string is left unchanged.

A, B	of type	SET
PA, PB	of type	SET_POINTER
Z	of type	LABEL : EXTENDED(EOF)

```
begin
    Put A into "read" mode ;
    Put B into "build" mode ;
    PA ⇐ A;  PB ⇐ B;
    Z ⇐ GETLABEL (PA);
    while Z ≠ EOF
        begin
            if Z = @ then
                PUTLABEL (@, PB)
            else if Z = 0 then
                PUTLABEL (1, PB)
            else
                PUTLABEL (0, PB);
            Z ⇐ GETLABEL (PA)
        end
end
```

8. Set predicates

The Boolean set predicates, i.e. set equality, element of, and set inclusion, can be evaluated from the basic operations FORMSET, UNION, INTERSECTION, and COMPLEMENT as follows:

1) A = B is *true* if and only if the label-list representations of A and B are identical, i.e. the lists are of the same length and have identical labels at each position.

2) $A \neq B$ is *true* if and only if there exists a label in the label-list representation of A that differs from the label in the corresponding position of the label-list representation of B.

3) $x \in A$ is *true* if and only if $\{x\} \cap A = \{x\}$, which can be evaluated using FORMSET and INTERSECTION.

4) $x \notin A$ is *true* if and only if $\{x\} \cap \neg A = \{x\}$, which can be evaluated using FORMSET, COMPLEMENT, and INTERSECTION.

5) $A \subseteq B$ is *true* if and only if $A \cap B = A$, which can be evaluated using INTERSECTION.

9. Cartesian products

Suppose X and Y are two discrete datatypes that have been used as the "base" sets, i.e. the Universes, for constructing two set algebra datatypes, SET OF X and SET OF Y, respectively, and suppose $A \subseteq X$ and $B \subseteq Y$ are instances of these datatypes. Since relationships, orderings, mappings, and operations on A and B can always be represented as subsets of Cartesian products, it is often useful to be able to represent the Cartesian product, A x B, or subsets of this product, in a format compatible with the representation of A and B themselves, i.e. as a list of the labels {0, @, 1}. We desire some integrated method for dealing efficiently with subsets of X x Y or A x B so that projections such as $\{ x \mid \langle x,y \rangle \in R \subseteq A \times B\}$ or other set theoretic operations can be processed by direct manipulation of the string representations without need for expansion to full display as elements of X, Y, or X x Y.

One method for representing an ordered pair $\langle x,y \rangle$ is to interleave the binary representations of x and y. For example, if $x = x_1,...,x_n$ and $y = y_1,...,y_n$, then $\langle x,y \rangle$ could be represented by the binary sequence $x_1,y_1,...,x_n,y_n$. Under this method, the length in bits of the binary sequence representing $\langle x,y \rangle$ is twice as long as the sequences representing x or y. The advantage of this approach is that the dyadic intervals defined in Section 3 of this paper are square or double-square regions in X x Y. It follows that contiguous, rectangular shaped regions in X x Y have efficient labeled tree representations.

Other methods for such representations, and efficient algorithms for set processing, are topics for future consideration.

Appendix

The purpose of this appendix is to define the mathematical terminology used in the text of this article. Sets and membership of sets are taken as primitives along with the axioms of set theory and the rules of logic. All structures are defined in terms of sets.

An **ordered pair** is denoted by $\langle x,y \rangle$ and is the two element set $\{\{x\},\{x,y\}\}$.

Given sets A and B the **Cartesian product** of A and B is the set:

$$A \times B = \{\langle x,y \rangle : x \in A \text{ and } y \in B\}$$

A **binary relation** on $A \times B$ is any subset of $A \times B$. The notation aRb denotes that $\langle a,b \rangle$ is an element of the binary relation R.

A **function** from A to B, denoted by $f:A \rightarrow B$ or by just f, is a binary relation on $A \times B$ satisfying:

Domain Coverage	1)	For every $x \in A$ there exists a $y \in B$ such that $\langle x,y \rangle \in f$.
Single-Valued	2)	If $\langle x,y \rangle \in f$ and $\langle x,z \rangle \in f$ then $y = z$. If $\langle a,b \rangle \in f$ then b is denoted by $f(a)$.

A **unary operation** on A is any function from A to A.

A **binary operation** on A is a function from $A \times A$ to A. If $u:A \rightarrow A$ is a unary operation, then the element u(a) is often denoted by ua or by a raised to the exponent u. If $*:AXA \rightarrow A$ is a binary operation, then $x*y$ denotes the element $*(\langle x,y \rangle)$.

A **Boolean algebra** is a tuple $(A,+,*,\neg)$ where A is a set containing distinguished elements 0 and 1, + and * are binary operations on A, and $\neg$ is a unary operation on A. A Boolean algebra must satisfy the following conditions for all x, y, and z in A:

Idempotent	1)	$x + x = x$	$x * x = x$
Associative	2)	$x+(y+z)=(x+y)+z$	$x*(y*z)=(x*y)*z$
Commutative	3)	$x+y = y+x$	$x*y = y*x$
Absorptive	4)	$x+(x*y)=x$	$x*(x+y)=x$
Distributive	5)	$x*(y+z)=(x*y)+(x*z)$	$x+(y*z)=(x+y)*(x+z)$
Null Element	6)	$x + 0 = x$	$x * 0 = 0$
Universe	7)	$x + 1 = 1$	$x * 1 = x$
Complement	8)	$x + \neg x = 1$	$x * \neg x = 0$

The **power set** of A is the set of all subsets of A. It is denoted by P(A). The power set of a set under set operations of union, intersection, and complement is a Boolean algebra with distinguished elements equal to A and the null set.

A **partial ordering** is a pair $(A, \leq)$ where A is a set and $\leq$ is a binary relation on A satisfying the following conditions for all x, y, and $z \in A$:

Reflexivity	1)	$x \leq x$
Antisymmetry	2)	$x \leq y$ and $y \leq x$ imply $x = y$
Transitivity	3)	$x \leq y$ and $y \leq z$ imply $x \leq z$

A **linear ordering** is a pair $(A, <)$ where A is a set and $<$ is a binary relation on A satisfying the following conditions for all x, y, and $z \in A$:

Trichotomy	1)	$x \neq y$ implies $x < y$ or $y < x$
Separability	2)	$x < y$ implies $x \neq y$
Transitivity	3)	$x < y$ and $y < z$ imply $x < z$

A **well ordering** is a linear ordering $(A, <)$ which satisfies the additional condition:

First Element	1)	For every non-empty subset $B \subseteq A$ there exists an $x \in B$ satisfying $x \neq y \Rightarrow x < y$ for all $y \in B$.

Every **finite linear ordering** is a well ordering.

In a partially ordered set $(A, \leq)$ the **predecessors** and the **descendants** of an element x are:

$$\text{Pred}(x) = \{ y : y \in A \text{ and } y \leq x \}$$
$$\text{Descd}(x) = \{ y : y \in A, x \neq y, \text{ and } x \leq y \}$$

A **tree** is a partial ordering with the additional property that the set of predecessors of any element is well ordered. That is, for a partial ordering $(T, \leq)$; if $<(z)$ is the set

$$\{<x,y>: x,y \in \text{Pred}(z), x \leq y, \text{ and } x \neq y \},$$

then $(T, \leq)$ is a tree if and only if $(\text{Pred}(z), <(z))$ is a well ordering for all $z \in T$.

Elements of a tree are called **nodes**.

Terminal nodes are nodes x for which Descd(x) is empty.

The **height** of a finite tree is the maximum cardinality of any predecessor set. That is:

$$\text{Height}(T) = \text{Max}\{ \text{Card}(\text{Pred}(x)) : x \in T \}$$

The **level number** of a node is denoted by Level(x), where $\text{Level}(x) = \text{Card}(\text{Pred}(x))$.

The **i-th level** of a tree is the set of all nodes having level number equal to i.

The **child descendants** of a node x in a tree T are defined as:

$$\text{Child}(x) = \{ y : x \leq y \text{ and } \text{Level}(y) = \text{Level}(x)+1 \}$$

Child descendants are often linearly ordered to distinguish relative position. This could be accomplished by maintaining an underlying linear ordering of T which induces a linear ordering on each Child(x).

A **binary tree** is a tree with the property that for every node x, either x is a terminal node or the set Child(x) has cardinality two. Elements of Child(x) in a binary tree are often distinguished as either Left or Right.

An **S-labeling** of a tree $(T, \leq)$ is a function $L_T: T \rightarrow S$ where S is a set of labels. The partial ordering $\leq$ of T extends to L_T as follows: If $t \leq t'$ in T then $\langle t,s \rangle \leq \langle t',s' \rangle$ in L_T. Under this induced partial ordering, $(L_T, \leq)$ is a tree with many of the same properties as $(T, \leq)$, and is thus defined to be the **labeled tree** corresponding to the labeling of T.

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4637

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
JULY 1991

4. TITLE AND SUBTITLE

Computer Implementation of a Discrete Set Algebra

5. AUTHOR(S)

Leonard Gallagher

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

Large finite sets occur naturally in computer data management. Discrete elements such as numbers, pointers, icons, and object identifiers all have fixed-length bit sequence representations that may be viewed as base-two integers. The efficient storage and manipulation of large collections of such items is a long-standing problem in computer science. In particular, many data management algorithms may be specified in terms of set operations (e.g. union, intersection, crossproduct) on these collections. Often a large set is stored in reduced form for storage efficiency. A specific problem is then to find efficient algorithms for performing set operations over the reduced representations. This paper presents a binary tree storage mechanism for efficient representation of arbitrary sets of discrete elements taken from a fixed universe. It then develops high-level algorithms for mapping these sets to and from their tree representations and for executing set operations directly on the storage representation itself. This approach provides a method for direct computer implementation of mathematical models based on Boolean algebras defined over finite sets.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

algorithm; binary tree; Boolean algebra; data management; data structure; labeled tree; mathematical model; partial ordering; set operations; set theory

13. AVAILABILITY

X	UNLIMITED
	FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).
	ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.
X	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
31

15. PRICE
A03

ELECTRONIC FORM