TECHNICAL REPORT NO. 324

Static Measures of Quadtree Representation of the Harwell-Boeing Sparse Matrix Collection

by

Peter H. Beckman

January 1991

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

Static Measures of Quadtree Representation of the Harwell-Boeing Sparse Matrix Collection

by

Peter H. Beckman

Computer Science Department
Indiana University
Bloomington, IN 47405

Abstract

The quadtree is an efficient data structure for uniform representation of both dense and sparse matrices. A quadtree matrix is either homogeneously zero, represented by the NIL pointer, or a 1×1 non-zero scalar, or a tree containing four subtrees, each recursively representing a quadrant of the matrix. This paper reports the results from experiments done on tools written in C to convert matrices from the **Harwell-Boeing Sparse Matrix Collection** to quadtree form. Results show that a preorder sequential traversal of a quadtree matrix provides efficient secondary storage and generally faster average access time to matrices in primary memory than row and column linked lists.

Section 1. Packed vectors and linked lists

Sparse matrices occur in a variety of scientific and engineering applications and special algorithms and data structures are used to reduce both the work and storage required to manipulate them. A suitable data structure reduces storage by compactly representing zeros and also provides a convenient way to avoid unnecessary computation on zeros. Two common representations for sparse matrices are packed vectors and linked lists [6]. *Packed vectors* store the non-zero entries as compressed vectors with corresponding indexes in another vector. *Linked lists* use records with a pointer field to a node containing the next non-zero element in the vector. Both methods reduce storage by compressing either the row or the column vectors of the matrix. For instance, a vector with 100 values, 80% of which are zero, could be represented by a compressed vector of 20

Research reported herein was supported by a DARPA/NASA Assistantship in Parallel Processing, number 26947K

floating point values, and 20 integer indexes. Such compression incurs a handicap; a packed vectors matrix cannot be easily traversed in the other orientation.

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & -3 & 6 & 0 & 0 & 0 & 0 \\ 0 & 18 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 5 & 10 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 15 & 9 & 0 \\ 0 & 60 & 11 & 0 & 0 & -5 & 8 & 6 \\ 8 & 82 & 52 & -8 & 16 & 10 & -2 & 1 \end{bmatrix}$$

Figure 1: An 8×8 Matrix

Array Subscript:	1	2	3	4	5	6	7	8	9
Row_Length:	2	2	3	2					
Row_Start_Index:	1	3	5	8					
Column_Index:	2	1	1	2	3	4	2	3	2
Matrix_Value:	2	1	4	7	-3	6	9	3	18

Figure 2: Rows 1-4 stored as packed vectors

Figure 1 is a simple 8×8 sparse matrix that is almost lower triangular. It will be used to illustrate these sparse matrix storage techniques. Figure 2 demonstrates the arrays storing its first four rows in packed form. It is common to leave the matrix values within a row unordered, since many algorithms expand the packed vector into a full vector before use. Algorithms using these structures are designed for entire row operations. To increase the speed with which these vectors are expanded and subsequently compressed, machines like the CRAY X-MP and the FACOM VP/400 have added special hardware and instructions to *GATHER* and *SCATTER* sparse vectors. [2, 4].

Figure 3 shows the logical representation of a linked list. In older versions of FORTRAN, **integer** arrays were used to link the lists. Modern FORTRANs provide pointer types. Figure 4 shows how FORTRAN arrays often implement a linked list.

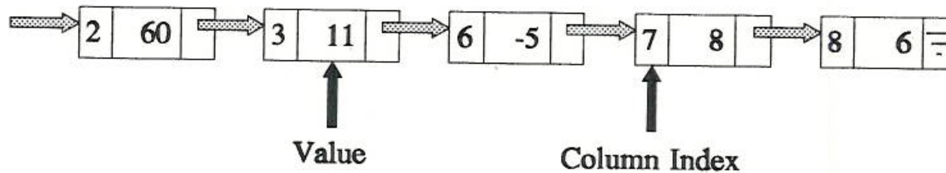

Figure 3: Logical representation of row 7 as a linked list

Array Subscript:	1	2	3	4	5	6	7	8	9	10	11
Row_Start_Link:	5	4	7	2							
Col_Index:	3	2		1	1	2	2		2	4	3
Forward_Link:	0	1	8	6	9	0	11		0	0	10
Matrix Value:	3	18		4	1	7	9		2	6	-3

Figure 4: Physical storage of rows 1-4 using FORTRAN arrays to form a linked list

For both packed vectors and linked lists, it can be difficult it can to traverse the data by columns when they are compressed by row. Two solutions are commonly used to provide faster access to the data by columns. One is to store two copies of the matrix, one copy using row compression, and the other, column compression. A more efficient use of space is to include links or indexes necessary for column traversal without duplicating the floating point matrix values [5].

If a linked list is expanded in this way, each non-zero element will be represented by a node with five fields: Matrix-value, Row-pointer, Column-pointer, Row-index, and Column-index. Figure 5 shows the logical representation of rows 1-4 linked by both rows and columns.

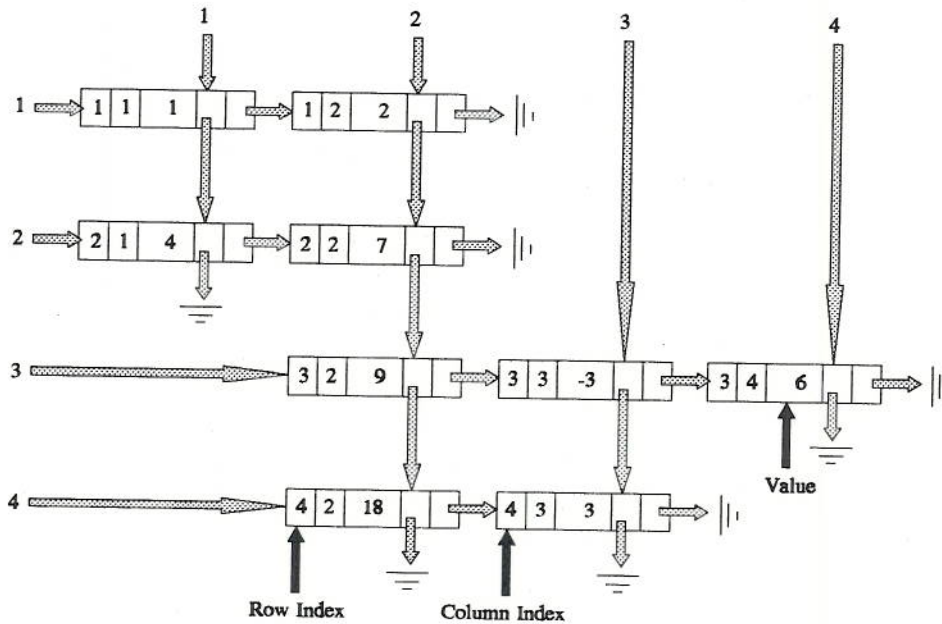

Figure 5: Rows 1-4 as a row and column linked list

Section 2. Quadtree representation

Using quadtrees is like using linked lists, but instead of packing the values by row and column, the data is packed by blocks (quadrants). However, row and column indexes are not stored in each node, since an element's position in the tree is sufficient to determine its coordinates. More formally, a *quadtree matrix* is represented in one of three ways. Either it is entirely *zero* and represented by the null pointer; or it is a non-zero scalar (a 1×1 matrix) represented by that scalar; or it can be cleaved into four equally-sized submatrices and is represented by a non-terminal node of four quadtrees [7]. When n is not a power of two, an $n \times n$ matrix is efficiently embedded in a $2^{\lceil \lg n \rceil} \times 2^{\lceil \lg n \rceil}$ representation by padding with NIL to the southeast (lower right).

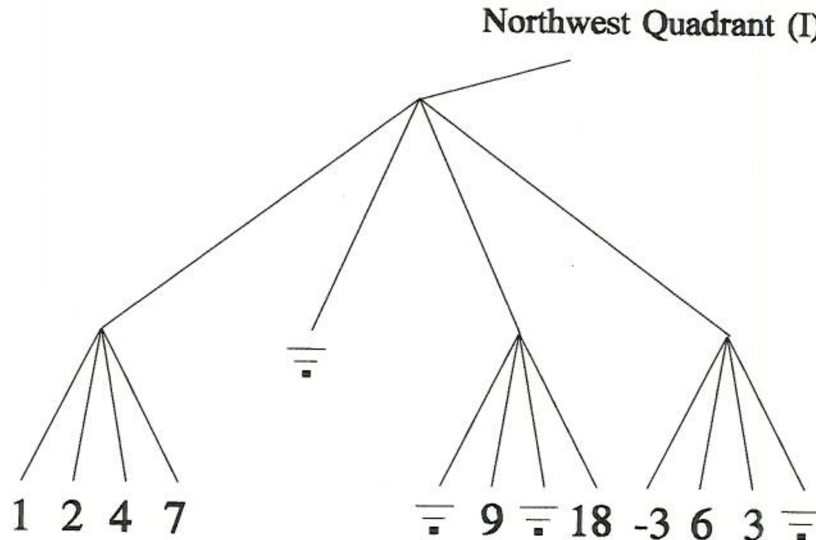

Figure 6: Northwest quadrant of Figure 1 in quadtree form

Recursive decomposition of the matrix in Figure 1 yields its quadtree decomposition. Its entire northeast quadrant is represented as a null pointer (NIL). Figure 6 shows how its northwest quadrant is represented in quadtree form. Physically, each interior node has four pointer fields, and a leaf node holds a matrix value, usually a floating point.

Section 3. The Harwell-Boeing sparse matrix collection

To provide researchers access to realistic examples of sparse matrices from a wide assortment of scientific fields, I. S. Duff et al. compiled the *Harwell-Boeing Sparse Matrix Collection*, a set of test problems [3]. The May 30, 1989, version contains 293 matrices represented by over 110 megabytes of ASCII data.

The Harwell-Boeing collection uses two methods to store sparse matrices. The specific details of the storage, which is based on FORTRAN input routines and an 80 column card-image format, are not needed for the experiments described

here. The two storage strategies represent two very different types of matrices: standard and elemental. Standard matrices are the most common, with 283 of the 293 matrices stored in this form. They result from a wide variety of applications. Elemental matrices, on the other hand, usually arise from finite-element applications, where numerous small elemental matrices are assembled to construct the final matrix. A specialized primary memory data structure for these matrices called clique storage is generally used [2]. Because specialized algorithms are used to handle these very special representations, they are not included in the experiments described later in this paper.

The conventional sparse matrix format uses a very simple storage scheme closely related to the packed vector data structure. The matrices are stored by columns, and defined by three lists: column pointers, row indexes, and values. For an $m \times n$ matrix, there are $n+1$ column pointers. The column pointers indicate the row indexes associated with each column. For every non-zero element there is one matrix value and one row index. The column pointers and row indexes define the sparsity pattern of the matrix. Some of the matrixes in the test set are given only as patterns to conserve space. A small header for each matrix provides other important information such as the number of rows and columns, title, properties, etc. One important property is whether the matrix is symmetric. If it is, only half of the matrix need be stored on disk. The set also provides for solution vectors and the right hand sides needed to solve linear systems, which were ignored in these experiments.

Section 4. Standard Sparse Matrix Format

The disk space necessary to store a sparse matrix in the Harwell-Boeing standard format is calculated quite easily. Letting *NNZ* be the number of non-zeros in the matrix and n be the number of columns,

$$size=(n+1)(sizeof(COLPTR))+(NNZ)(sizeof(ROWIND)+sizeof(VALUE))$$

predicts the *raw* storage required. The matrices in the Harwell-Boeing collection are stored in ASCII or EBCDIC to ensure the highest degree of cross-manufacturer compatibility. The term 'raw' in computer jargon refers to unprocessed binary data, and in this case, is used to denote that the matrix is being held a in binary, machine dependant form. The *sizeof* function computes how many bytes are needed to represent a data type. For all the matrices in the Harwell-Boeing collection, a standard two byte unsigned integer can be used to hold a row index or a column pointer.

ASCII:	25.4%
RAW:	23.7%

Table I: Average pattern portion of matrix in Harwell-Boeing standard sparse format

Since all representations must contain the matrix values, potential savings in space come from the pattern definition. The first experiment was run to determine how much disk storage was devoted to the pattern structure (column pointers and row indexes) of a matrix compared with the total space. In ASCII card-image format, an average of 25.4% of a matrix's total space was devoted to column and row data. Of course only the 221 non-pattern matrices from the collection could be used for this comparison. Unfortunately that figure is somewhat misleading; FORTRAN formatting permits a wide range of efficiency, with plenty of "white space" usually included. The storage needed for ASCII representation of an integer is usually six to eight bytes, compared to a two-byte raw representation. On the other hand, an ASCII FORTRAN floating point value, with a decimal and exponent, can use 16-26 bytes compared to a raw eight byte IEEE double

precision floating point representation. It is clear that a lot of variance can occur. If the ratio of pattern structure to total storage is calculated using raw storage, with column pointers and row indexes two bytes each and matrix values eight bytes, an average of 23.7% of a matrix's raw storage is the pattern structure.

Section 5. Preorder sequential storage

A matrix held in quadtree form in main memory, can be threaded a number of different ways to produce an ordering that can be written sequentially to disk. One convenient method is a preorder sequential traversal [5]. The tree is descended recursively, in order, from quadrant I to quadrant IV. The Appendix shows two extremely simple, recursive, C programs that can be used to read and write quadtrees to disk. Only four bits of information are needed at each interior (non-leaf) node, to describe it; each bit corresponds to whether a quadrant is completely zero. When the bottom of the tree is reached, the value in the leaf node can be written to disk. For ease of use, the four bits of pattern information are byte justified, with the high order four bits remaining unused. Symmetric matrices, like matrices in standard sparse format, only need to store half of the matrix.

This is an extremely compact way to represent the sparsity pattern of a matrix. As mentioned earlier, the actual number of matrix values written to disk in both the Harwell-Boeing standard sparse format and preorder sequential traversal of a quadtree are the same. Any savings is incorporated in an improved pattern coding. Figure 7 illustrates a preorder traversal on the northwest quadrant of the Figure 1 matrix. Figure 8 compares several different storage methods. The pattern information used two-byte unsigned integers for the column pointers and row indexes. Unix **COMPRESS** was applied to each file to show how much storage would be used in a real programming environments.

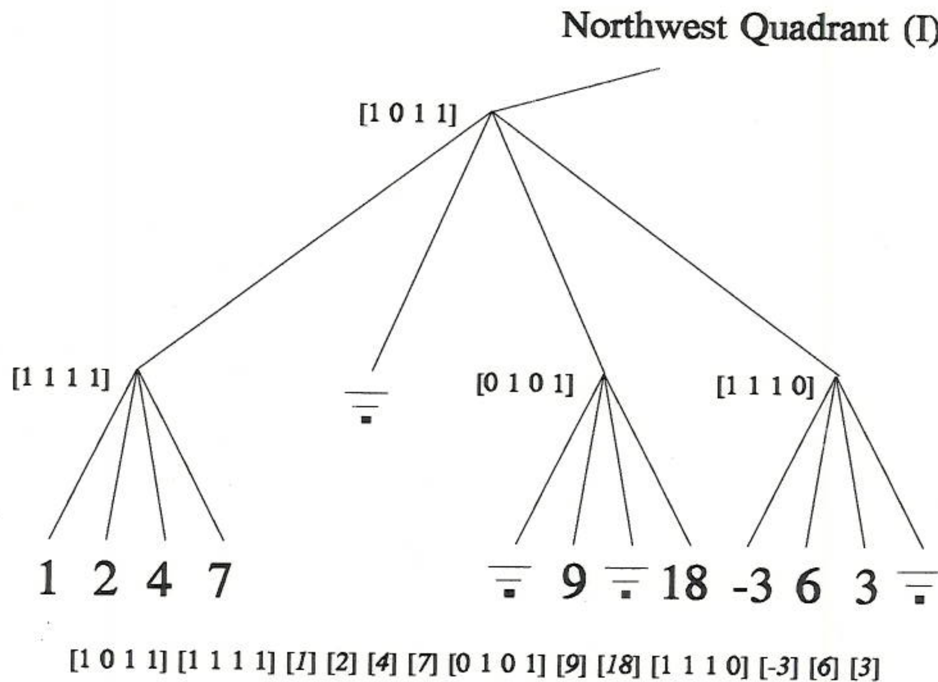

Figure 7: A Preorder Sequential Traversal

Section 6. Accessing matrix elements in primary storage

Each data structure used to store sparse matrices in primary memory has its own advantages and disadvantages. It is beyond the scope of this paper comprehensively to compare and to contrast them. Readers interested in learning more about quadtree matrix storage and algorithms are referred to papers published on that subject by Wise [7, 8]. This paper considers two very simple measures of efficiency: memory consumption and average element access time. The quadtree storage of the Harwell-Boeing matrices will be compared to a row and column-linked list. The reason for this comparison, rather than comparing quadtrees to packed vectors, or quadtrees to row linked lists, is that the properties of quadtrees most closely resemble that of row and column linked lists. Both use memory nodes linked together with pointer fields, and both are traversable by column or row with the same ease.

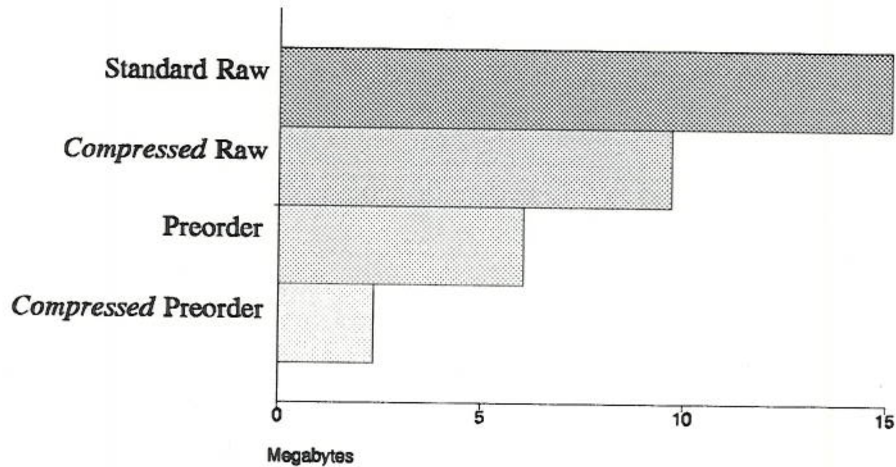

Figure 8: Data required to store the pattern portions of the 283 standard sparse matrices from the Harwell-Boeing collection.

As described earlier, each linked list node has five fields. At any given moment during the computation, the total storage for an $m \times n$ matrix is

$$space = (n+m)(sizeof(pointer)) + NNZ(sizeof(value)) + 2 \times sizeof(index) + 2 \times sizeof(pointer))$$

The column and row indexes can comfortably be two-byte unsigned integers, and in most modern programming environments, pointer fields are at least 32 bits wide. This means that each linked list node will have 12 bytes of pointer and index data, plus whatever bytes are necessary to store the matrix value. Two pointer arrays will also be needed to store the first link for each row and each column. This structure is fairly compact for very sparse matrices, but the storage grows linearly with the number of non-zero values.

An interior node is comprised of four pointer fields, one to each quadrant. A leaf node only has a value field. For all but the most sparse matrices, the quadtree

data structure uses less space than the linked list representations. Adding non-zero values to a linked list will always require the same amount of overhead; four fields per node. Adding a non-zero to a quadtree, in the worst case, generates *log(n)* interior nodes of overhead. But as more and more non-zero values are added to a quadtree, less and less overhead is added because interior nodes are shared. In the best case, no overhead is created, because all interior nodes are already in place. This is a very important feature; solving systems of equations and other matrix operations cause many of the zeros to fill-in [2]. While quadtrees add less overhead as the matrix fills in, linked lists keep adding a constant amount. Yet, averaging across all 283 standard but very sparse Harwell-Boeing matrices, quadtree storage used only about eight percent more than linked lists.

The contrast between trees and lists is also apparent when determining the number of memory reads necessary to find an element. For an $n \times n$ quadtree, it takes at most

$$\lceil \log_2 n \rceil + 1$$

reads to locate a matrix value [8]. This contrasts to the reads required to find a

$$2n + 2$$

matrix element in a linked list under the worst case.

Analogous with space, this overhead can only worsen as the matrix fills in, as linked list chains become longer. To compare the number of reads necessary to find a random (i,j) element for each of the 283 matrices, an average was calculated for each matrix.

For most matrices, the number of reads for each format was very close to the other. However, matrices that are not very dense, create long linked lists. Figure 9 illustrates one such matrix. These matrices do quite poorly; Figure 10

demonstrates how the data is skewed by the relatively few dense matrices included in the Harwell-Boeing test set.

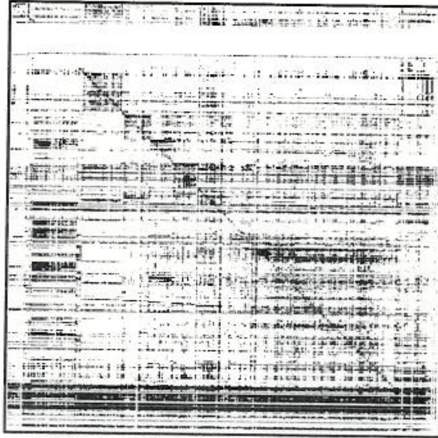

Figure 9: 496×496 Matrix
H-B Key: MBEACXC

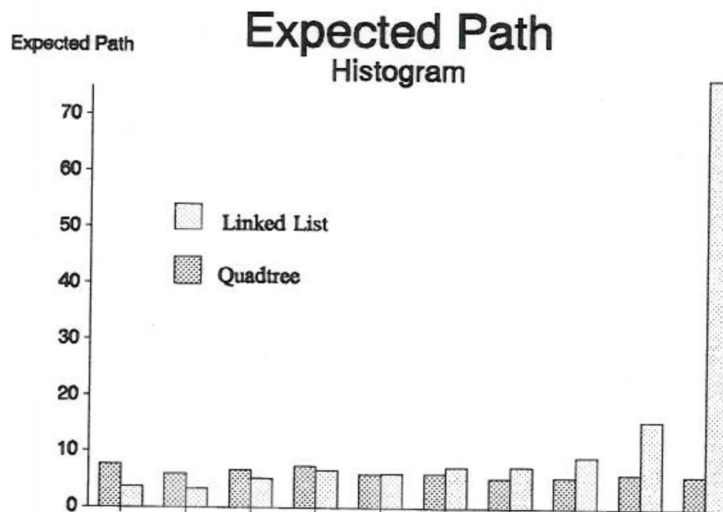

Figure 10: Average memory reads to find random (i,j) element in matrix

Section 7. Conclusions

Storing static matrices as quadtrees is efficient for both primary and secondary storage. The space savings achieved with preorder storage occurs only from that portion of the data describing the sparsity pattern of the matrix. In all storage schemes, the number of matrix values stored remains constant. A matrix can also be decomposed into randomly accessed blocks, each of which written to disk in preorder form. The compression permits disk pages to hold more data than conventional methods.

Another advantage is the relative simplicity of the C code needed to read or write a matrix (Appendix). Only the depth of the tree (size of the matrix) is needed; the preorder threading is managed by the C control stack, and does not burden the programmer.

Moreover, the quadtree data structure grows gracefully in size, both for disk and memory storage. This grace makes it more flexible than linearly linked structures, capable of supporting algorithms and data that generate different degrees of sparsity [1].

The most important feature of quadtree storage is how easily divide-and-conquer algorithms can be applied. Block algorithms permit easy decomposition of complex tasks for scheduling onto parallel processors. This paper presents only static performance measures for quadtrees; future work [1] will examine the dynamic properties of quadtree representation during LU decomposition on an MIMD computer. Also of interest is the underlying operating system that supports heap based multiprocessing.

APPENDIX

C procedures for reading and writing matrices in preorder form

```
#define NWbit 8
#define NEbit 4
#define SWbit 2
#define SEbit 1
/* ********** Read in preorder matrix ********** */
mem_node *read_preorder(depth)
  int depth;
  {
  unsigned char path;      /* one byte, pattern portion of data */
  mem_node *rp;            /* return pointer to memory node */

  rp = getnode();          /* Get an empty memory node */
  if (depth == 0)          /* At bottom, time to read value */
    { fread(&(rp -> value),sizeof(double),1,inputstream);
      return(rp); }
  else
    {
    fread(&path,sizeof(unsigned char),1,inputstream);

    if ((path & NWbit) > 0)  /* NON-NULL NorthWest Quadrant */
      rp -> NorthWest = read_preorder(depth-1);
    else rp -> NorthWest = 0;
    if ((path & NEbit) > 0)  /* NON-NULL NorthEast Quadrant */
      rp -> NorthEast = read_preorder(depth-1);
    else rp -> NorthEast = 0;
    if ((path & SWbit) > 0)  /* NON-NULL SouthWest Quadrant */
      rp -> SouthWest = read_preorder(depth-1);
    else rp -> SouthWest = 0;
    if ((path & 1) > SEbit)  /* NON-NULL SouthEast Quadrant */
      rp -> SouthEast = read_preorder(depth-1);
    else rp -> SouthEast = 0;

    return(rp);
    }
  }
```

```
/* ********** Write preorder matrix ********** */
void write_preorder(x,depth)
  mem_node *x;
  int depth;
  {
  unsigned char i;

  if (depth != 0)        /* currently visiting interior node */
    i = 0;
    if (x -> NorthWest != 0) i = i | NWbit;
    if (x -> NorthEast != 0) i = i | NEbit;
    if (x -> SouthWest != 0) i = i | SWbit;
    if (x -> SouthEast != 0) i = i | SEbit;

    fwrite(&i,sizeof(unsigned char),1,outputstream);

    if (x -> NorthWest != 0)
       write_preorder(x -> NorthWest, depth-1);
    if (x -> NorthEast != 0)
       write_preorder(x -> NorthEast, depth-1);
    if (x -> SouthWest != 0)
       write_preorder(x -> SouthWest, depth-1);
    if (x -> SouthEast != 0)
       write_preorder(x -> SouthEast, depth-1);
    }
 else  /* at leaf node */
    fwrite(&(x -> value),sizeof(double),1,outputstream);

 }
```

Bibliography

[1] Beckman, P. H. *Ph.D. Dissertation* (in progress). Computer Science Department, Indiana University

[2] Duff, I. S., Erisman, A. M., and Reid, J. K. *Direct Methods for Sparse Matrices*. Oxford University Press, New York, 1989

[3] Duff, I. S., Grimes, R. G., and Lewis, J. G. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15 (March 1989), 1-14

[4] Hwang, K., and Briggs, F. A. *Computer Arcitecture and Parallel Processing*. McGraw-Hill, New York, 1984.

[5] Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundemental Algorithms.* Addison Wesley, Reading Mass. 1975, Section 2.2.6 and 2.3.3

[6] Osterby, O., and Zlatev, Z. *Direct Methods for Sparse Matrices*. Springer-Verlag, New York, 1983.

[7] Wise, D. S. Representing Matrices as Quadtrees for Parallel Processors (extended abstract). *ACM SIGSAM Bulletin 18*, 3 (Aug 1984), 24-25.

[8] Wise, D. S., Franco J. Costs of Quadtree Representation of Non Dense Matrices. *Journal of Parallel and Distributed Computing*, 9 (1990), 282-296.