OFF PRINT FROM

DESIGN AND IMPLEMENTATION OF OPTIMIZATION SOFTWARE

edited by

HARVEY J. GREENBERG
Department of Energy, Washington, D.C., U.S.A.

SIJTHOFF & NOORDHOFF 1978
Alphen aan den Rijn – The Netherlands
A TUTORIAL ON MATRICIAL PACKING

HARVEY J. GREENBERG

Energy Information Administration
Washington, D.C. 20461

ABSTRACT. This tutorial describes advanced data structures for representing matrices, as implemented in many (large-scale) mathematical programming systems. It functionally separates the "endogenous matrix file" from the "exogenous matrix file." The exploitation of sparsity and super-sparsity is the primary foundation.

This tutorial describes fundamental data structures to represent a large matrix. The technology has been developed from linear programming systems, but it can be applied directly to any "matricial program." A matricial program is one which can be represented by a matrix (or more formally a tensor). An example of a class of nonlinear matricial programs is the multinomial forms embodied in geometric programming. Further, the data structures we shall describe may be extended in several ways to represent more general nonlinear programs.

When we say a matrix is large, we mean it has at least 1,500 rows and 10,000 columns. Clearly, we would not want to use a simple array structure and reserve 15,000,000 storage locations. Even small-to-medium size matrices are of the order 800x5,000, so we need something more sophisticated than an array representation.

Fundamental to the development of matricial packing is understanding what a matrix is. To a mathematician a matrix is a linear transformation from one vector space into another; to an engineer it is a rectangular array of numbers. The view we shall take in our computer science orientation is that a matrix is a
"file", that is a collection of records with a prescribed structure.

Before we consider the types of matrix files relevant to the design of a mathematical programming system, let us note that real models produce matrices where most of the matrix elements are zero. This is an important observation because we can avoid storing zeroes by designing the records to identify only non-zeroes. Arithmetic operations, such as addition or multiplication, have predictable results for the zeroes, and they may be avoided, thus resulting in a savings of time as well as space. Retrieval operations, such as determining a matrix element value for given row and column specification, can begin with zero as a default and search for a nonzero, terminating with the appropriate conclusion.

When a matrix has a high percentage of zeroes, it is called "sparse," and when it has few zeroes, it is called "dense." More precisely, we have the following

**DEFINITION:** The sparsity of an m by n matrix is the ratio of the number of zero elements to the product, m*n. The density is 1 - sparsity.

Small models, say with 300-800 rows and 500-1200 columns, tend to have a density of approximately .5 - 2%; medium models, say with 800-1500 rows and 1200-7000 columns tend to have a density of approximately .05 - .3%; very large models, say more than 2500 rows and more than 30,000 columns tend to have less than .01% density. It is generally true that the larger the model the lower the density, which is one important reason we can represent them for effective processing.

To gain some insight as to why this is true, consider the standard transportation model with s sources and d destinations. The matrix size is (s+d+1) rows and s*d columns (if every link is present) and every column has exactly two ones in it for the supply-demand constraints plus a cost of flow in the objective row. Therefore the density is less than 3/(s+d+1), which decreases with increases in the problem dimensions.

More generally, a modeller cannot define activities with more than a few row elements; in practice, he would tend to use more activities rather than fewer complicated ones. This results in the rule-of-thumb: a column will typically have 5 non-zeroes, regardless of the number of rows. This seems to be true with a small variance. Thus, if we assume each column count is less than some constant, c, independent of row size (where c is about 7), then
$$\text{density} \leq c^* n = c/m.$$  

Thus, as the number of rows gets large, the density decreases.

2. ENDOGENOUS MATRIX FILE

2.1 Basic Structures

Let us now consider the types of matrix files which are relevant to the design of a mathematical programming system. The one we shall consider first is the endogenous matrix file. This is to be constructed for use during iterations with the primal simplex method, and it represents a single problem.

Under the rules of the primal simplex method the only access to the endogenous matrix file is for a given column; in many designs one simply proceeds column by column in a sequential fashion to find one which corresponds to an activity that shows promise of improving the objective function. Thus, let us suppose that the basic type of access and operation to be performed is characterized by:

1. access by column, and
2. compute inner product of a column with another vector, stored as a linear list.

The access by column suggests that each record represents the activity's column, including any relevant values such as scale, translation or bound values. Therefore, we define columnar form to mean that record j corresponds to activity j. All of the schemes we shall consider for the endogenous matrix file will be of columnar form. There is an obvious symmetry in developing a row form representation.

Storage as an array results in a linear list with the location (cr address) given by the following linear equation:

$$\text{LOC}(A_{ij}) = \text{LOC}(A_{00}) + cnj + ci$$

where

- $n$ = number of columns + extra information (e.g., scale, bounds).
- $c$ = number of addressable units to store a value.

The units are number of addressable entities, (e.g., bytes or
words), using high precision (e.g., word on CDC 6000's, double word on IBM 370's or UNIVAC 1108).

Since we know most models are sparse, let us consider more compact data structures, resulting in nonlinear lists with variable record sizes, depending upon the density.

GENERAL COLUMNAR FORM:

The j-th record corresponds to the j-th activity and may include a "header" to provide record separation plus other information about the activity.

Figure 1: Schematic of Columnar Form

header ............

extra information...

record for j-th activity

column A_j:...........

We shall consider the header and the extra information later. Our concern now is with the representation of the column other than the array form.

Let us begin with an elementary scheme,

BASIC SEQUENTIAL COLUMNAR FORM (BSC):

Each data item for the column in the j-th record corresponds to a nonzero element and appears as the couple: (VALUE, ROW INDEX).

In order to illustrate BSC let us assume, temporarily, that no extra information is needed, and the header simply contains the number of nonzeroes in the column. Then, consider the following matrix:
The endogenous matrix file for this matrix will have 4 records, as illustrated in Figure 2 below:

Figure 2: Schematic of Example BSC

```

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0, 0</td>
</tr>
<tr>
<td>1.0, 1</td>
</tr>
<tr>
<td>-1.0, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0,0</td>
</tr>
<tr>
<td>-1.0,1</td>
</tr>
<tr>
<td>2.0,2</td>
</tr>
<tr>
<td>1.0,3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0,0</td>
</tr>
<tr>
<td>2.0,4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0,1</td>
</tr>
<tr>
<td>1.0,4</td>
</tr>
</tbody>
</table>
```

Let us now consider how to perform an inner production using BSC. Let \( X(1), \ldots, X(m) \) be an array (i.e., linear list representation of a vector). At the risk of seeming trivial, notice that

\[
(X, A_j) = \sum_{i=1}^{m} A_{ij} * X(i) = \sum_{i:A_{ij} \neq 0} A_{ij} * X(i),
\]
so only non-zeroes are needed in computing the inner product. Further, the order of summation is of no (theoretical) consequence, so the row indices stored in BSC format need not be in sort order.

**Algorithm BSCPROD:** Given an array, \( X(1), \ldots, X(m) \), and a column in BSC format with the \( k \)-th data item of the form \((v_k, i_k)\), where \( 1 \leq i_k \leq m \), this algorithm computes the inner product, \((X, A_j)\), and puts the result into \( PROD \).

- **Step 1 (Initialize):** Set \( PROD = 0 \), set \( k = 1 \) and perform Step 2 until end-of-record is reached.

- **Step 2 (Add Term):** Retrieve \((v_k, i_k)\) and set \( PROD \leftarrow PROD + v_k \times X(i_k) \); advance \( k \).

Of course, since we may compute the inner product for more than one column in the endogenous matrix file, an exogenous pointer is maintained to locate the column’s header; algorithm BSCPROD simply shows the elementary scheme for one column. It should be noted that because we inherently avoid the multiplication and addition (Step 2) for the zero elements, the BSC form takes less time than the array form to compute an inner product. Now let us consider storage requirements.

For an array the storage is \( mn \) locations for the matrix elements plus whatever is needed for the extra information. In BSC we incur the extra storage for the headers and the row indices. We have, for BSC,

\[
\text{STORAGE} = n \times (h + x) + (v + i) \times Z,
\]

where

- \( n \) = number of columns
- \( h \) = size of header
- \( x \) = size of extra information
- \( v \) = size of non-zero value
- \( i \) = size of index value
- \( Z \) = number of non-zeroes

Then, the ratio of BSC storage to the array storage is,

\[
R = \frac{(h + x)}{(m + x)} + \frac{(v + i) \times \text{DENSITY}}{(1 + x / m)}.
\]

The first term is relatively small, especially for large row size; the second term becomes small when the density is low. For definiteness we may take the word (say on CDC 6000's) as a basic
storage unit in the array. Then consider h=1, x=0 and v+i= 1.25. This yields a storage ratio,

\[ R = \frac{1}{m} + 1.25 \times \text{DENSITY}. \]

For large m the BSC form takes less space when DENSITY < .80. Since we typically have DENSITY << .50 (often less than .05), the BSC form generally takes much less space than an array representation of a matrix. Further, by sacrificing some precision we can take v+i=1, in which case only the headers contribute extra storage, and this is negligible even for high density.

Now let us consider another scheme based upon the observation that most models have a prevalence of plus or minus ones. If we could distinguish them, then we could save time by avoiding multiplication during the computation of an inner product; further, if the data structure distinguishes them, then we do not have to store their values. This leads to the

PARTITIONED SEQUENTIAL COLUMNAR FORM (PSC)

The data items for the column in the j-th record is partitioned into 3 sections with the following structure:

section 1: row index list associated with i: A_{ij}=1

section 2: row index list associated with i: A_{ij}=-1

section 3: BSC form for i:A_{ij}≠0,1,-1.

Using our former example let the header contain the length of each of the three sections. Then, we have a structure illustrated in Figure 3.

ALGORITHM PSCP: Given an array, X(1), ..., X(m), and a column in PSC, this algorithm computes the inner product, (X,A_j) and puts the result into PROD.

step 1 (initialize): set PROD = 0.

step 2 (+1 section): for each row index, i, in the +1 section, set PROD + PROD + X(i).

step 3 (-1 section): for each row index, i, in the -1 section, set PROD + PROD - X(i).

step 4 (BSC section): for each data item, (v,i), in the BSC section, set PROD + v*X(i).
Notice that we save the multiplication for the ones in the "ones sections" (steps 2 and 3), but there is extra "overhead" in performing the loops, compared to algorithm BSCPROD. An exact comparison would depend on the particular machine. It should also be noted that the +1's may be combined in one section by placing the sign on the row index. The tradeoff is in the loop overhead in algorithm PSCPROD. If there is only one section for ones, then the sign would be tested for each row index in order to use

$$\text{PROD} + \text{PROD} + X(i)$$

or

$$\text{PROD} + \text{PROD} - X(-1)$$

in the accumulation step over the ones section. If the number of ones is "small," the combined section is better because only one loop is used. Conversely, if there is a "large" number of ones, the extra loop is better to avoid a comparison at each step. The threshold is machine dependant. (For IBM 370's it is about 4 to 5, making the combined section typically preferrable.)
Now consider the storage requirements for PSC. We have

\[ \text{STORAGE} = n(H+x) + Z + g, \]

where

- \( n \) = number of columns
- \( H \) = header size
- \( x \) = size for extra information
- \( Z \) = number of nonzeroes
- \( i \) = size of index
- \( g \) = number of general nonzeroes (\#0,1,-1)
- \( v \) = size of value

Compared to BSC we save \((Z-g) v- (H-h)n\), that is the number of ones (+ or -) times the size of general values, plus a possible difference in header space. Typically, ones are very prevalent and account for a high percentage of the density. Further, the difference in header space is relatively negligible in large models. Thus, PSC storage requirements are much less, as a rule.

Now let us consider an alteration in the BSC format based upon the observation that many nonzero values appear many times throughout the matrix, even if we do not count the ones (which could be represented with PSC form). For reasons that become clear only when studying many models and how they are generated, the number of distinct constants is much less the number of nonzeroes. Perhaps one type of model where this is evident is one involving time periods, where the same submatrix is replicated because those sets of relations exist in every time period. Another explanation is that the "raw data" is relatively small compared to the matrix which gets generated from this raw data; thus, it is "likely" that the same number will appear in many places. Of course, highly structured models, such as ones with a large embedded network are inherently supersparse with an abundance of +1.

Whatever the reason, it is generally the case that many nonzero values are replicated. If we assume that it takes less space to store an index (some pointer) than a nonzero value (in high precision), then we may construct an "inverted file" structure for the endogenous matrix file. This is the same technique used in compiler design, where we setup a literal pool, that is a table of constants. Then, instead of \((v, i)\) containing the value and row index as in BSC, we let \( v \) be an index to identify the location in the literal pool whose contents is the nonzero value. This leads to the following scheme.
INVERTED SEQUENTIAL COLUMNAR FORM (ISC):

The column in the j-th record has one data item per nonzero in the form: (VALUE INDEX, ROW INDEX), where the value index locates the nonzero value in a LITERAL POOL.

Of course, we may combine PSC with ISC by using ISC form in section 3 (instead of BSC). However, let us illustrate ISC with our example and not use PSC form.

Figure 4: Schematic of Example ISC.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1,0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3,3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4,0</td>
<td>3,1</td>
<td>5,2</td>
</tr>
<tr>
<td>2,3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2,0</td>
<td>5,4</td>
<td></td>
</tr>
<tr>
<td>2,1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...header for column 1

LITERAL POOL

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.0</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>-1.0</td>
</tr>
<tr>
<td>4</td>
<td>3.0</td>
</tr>
<tr>
<td>5</td>
<td>2.0</td>
</tr>
</tbody>
</table>

...header for column 2

...header for column 3

...header for column 4

We shall consider an inner product algorithm for the ISC format, and extension to combine with PSC should be straightforward.

ALGORITHM ISCPROD: Given an array, X(1),...,X(m), and a column in ISC, this algorithm computes the inner product, (X,Aj), and puts the results into PROD.

step 1 (initialize): set PROD = 0 and perform step 2 until an end-of-record is reached.

step 2 (add term): retrieve (v,i) data item, and set V=VALUE(v) (from the literal pool of values), set PROD = PROD + V*X(i).

We see that there is an extra retrieval in step 2 compared with algorithm BSCPROD, so algorithm ISCPROD will generally be slower. However, let us consider the storage requirements.
We have

\[
\text{STORAGE} = n \times (h+x) + 2 \times Z \times i + v \times V,
\]

where

- \( n \) = number of columns
- \( h \) = header size
- \( x \) = size for extra information
- \( Z \) = number of nonzeros
- \( i \) = size of index
- \( v \) = size of value
- \( V \) = number of distinct nonzeros

The difference between ISC and BSC is \( Z \times i + v \times (V-Z) \). Dividing this by \( Z \times v \) we consider \( i/v + V/Z - 1 \). Therefore, if the ratio of distinct nonzeros to the total number of nonzeros \( (V/Z) \) is sufficiently small, then ISC takes less space (including the storage for the literal pool) than BSC. For definiteness consider \( i/v = 1/4 \); then we may conclude ISC uses less space if \( V/Z < 3/4 \). It is generally true that \( V/Z < 1/2 \), so ISC typically takes much less space than BSC.

If we accept the claim that ISC form uses much less space than BSC, and we combine it with the space-saving PSC form, then we may be able to avoid costly I/O by keeping the entire endogenous matrix file in core. This may more than compensate for the time lost in step 2 of algorithm ISCPROD.

Now let us consider how to establish the literal pool. When a value is received as input to be added to the endogenous matrix file, we must search the literal pool to see if it is there. If so, then we want the index number to insert into the associated data item; if not, then we want to add it as a new entry to the literal pool and insert it into the associated data item. A technique for doing this efficiently is known as "hashing."

Hashing is the process of computing an address (or index) from an input key (the nonzero value in our case). This process is carried out with the use of a "hash function" which is the associated mapping from the set of input keys to the set of addresses (or indices). We shall not take up the entire subject of hashing here, but we shall describe one such process suitable for our needs.

Let \( h \) be a function which has domain equal to the representation of a floating point value, which is our input. Its range shall be the integers from 1 to \( L \), where \( L \) is to be prescribed. One example is to let \( L=2^k \), so a string of \( k \) bits is a binary representation of \( h(v) \). Now partition the input, \( v \), into fields, each of length \( k \). Treating these fields as a separate binary string we can use arithmetic or boolean operations to form \( h(v) \). For definiteness, consider \( k=8 \) (e.g., byte on IBM 370's), and let \( v \) be represented by 64 bits (double word on IBM 370's). Then,
there are 8 fields. We can take the exclusive-OR operation among the 8 fields to obtain \( h(v) \). Here is a list of hash values for this particular function with the representation found on IBM 370's:

<table>
<thead>
<tr>
<th>Value</th>
<th>Hecadecimal Representation</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>4110 0000 0000 0000</td>
<td>81</td>
</tr>
<tr>
<td>-1.0</td>
<td>C110 0000 0000 0000</td>
<td>209</td>
</tr>
<tr>
<td>10.0</td>
<td>41A0 0000 0000 0000</td>
<td>225</td>
</tr>
<tr>
<td>-10.0</td>
<td>C1A0 0000 0000 0000</td>
<td>97</td>
</tr>
<tr>
<td>2.0</td>
<td>4120 0000 0000 0000</td>
<td>97</td>
</tr>
<tr>
<td>0.5</td>
<td>4080 0000 0000 0000</td>
<td>192</td>
</tr>
<tr>
<td>0.333</td>
<td>C055 5326 17C1 BDA5</td>
<td>46</td>
</tr>
<tr>
<td>8.0</td>
<td>4180 0000 0000 0000</td>
<td>193</td>
</tr>
<tr>
<td>17.0</td>
<td>4211 0000 0000 0000</td>
<td>83</td>
</tr>
<tr>
<td>0.99</td>
<td>40FD 70A3 D70A 3D70</td>
<td>254</td>
</tr>
<tr>
<td>-2.0</td>
<td>C120 0000 0000 0000</td>
<td>225</td>
</tr>
</tbody>
</table>

To illustrate, consider \( h(1.0) \). Expanding the hexadecimal representation into binary form we have 01000001 00010000 0...0, so that the exclusive-OR among the eight fields (viz., bytes) yields the 8-bit string, 01010001. This is the binary representation of 81.

Given a hash function, such as the one just described, let the hash value be a list number, pointing to a linear list called HEADS. If \( h(v) = i \), then we look at HEADS(i). If it is zero, then this means the value, \( v \), has not appeared before, and we create a list by adding \( v \) to the literal pool and setting HEADS(i) to point to it (as the first member of this list). If HEADS(i) is not zero, then it points to the literal pool, and we begin a search through a linked list starting at this first member. If \( v \) is found, our task is complete upon noting the associated value index; otherwise, \( v \) is added to the literal pool and the associated list is updated with this new insertion. This procedure is given by the following algorithm (to be applied to one entering value).

**ALGORITHM ENTERV:** Given the input value, \( v \), and the associated hash value, \( h(v) \), the value index is obtained and put into INDEX. In the event no space remains and \( v \) must be inserted, OVERFLOW is reached. The following information is included:
HEADS...linear list of list heads whose length is the range of the hash function, h;

POOL....literal pool of constants between location 1 and MAX;

LINK....parallel list to POOL containing links for (multiple) value lists;

LAST....pointer to last member of POOL.

TOLERANCE...Nonnegative value.
Initially (before first entry) all of the above information (except TOLERANCE) is zero.

step 1 (Start with list head): set INDEX=HEADS(h(v)).

step 2 (Is list empty?): if INDEX=0, go to step 4; else, if / POOL(INDEX)-v / ≤ TOLERANCE, STOP; else, set j=LINK(INDEX).

step 3 (Does the list end here?): if j=0, go to step 4; else set INDEX=j, and go to step 2.

step 4 (Add entry.): if LAST=MAX, exit OVERFLOW; else, advance LAST ← LAST+1, set LINK(LAST)=INDEX, HEADS (h(v))=LAST, POOL(LAST)=v and INDEX=LAST. EXIT.

If we apply algorithm ENTERV to each of the values listed in our example, then we obtain the following literal pool:

<table>
<thead>
<tr>
<th>HEADS LOC</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>7</td>
</tr>
<tr>
<td>81</td>
<td>1</td>
</tr>
<tr>
<td>83</td>
<td>9</td>
</tr>
<tr>
<td>97</td>
<td>5</td>
</tr>
<tr>
<td>192</td>
<td>6</td>
</tr>
<tr>
<td>193</td>
<td>8</td>
</tr>
<tr>
<td>209</td>
<td>2</td>
</tr>
<tr>
<td>225</td>
<td>11</td>
</tr>
<tr>
<td>254</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LITERAL POOL LOC</th>
<th>VALUE</th>
<th>LINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-1.0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>10.0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>-10.0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0.3333</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>17.0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0.99</td>
<td>0</td>
</tr>
</tbody>
</table>
Notice in this example that both -10.0 and 2.0 hash to the same value (i.e., 97). This is known as a "collision," and these values are in the same list. For this example there are no other collisions, so the other lists have at most one entry. If we suppose that the hash function distributes uniformly over its range, then we will have R lists, where R is the range of h. In our particular choice of hash function, there are 256 lists. Thus, if, for example, we have 25,600 distinct nonzero entries, each list will have about 100 entries. The average search will be equivalent to a linear search through a linked list of 50 elements. If we are to use direct hashing (i.e., without a HEADS list), then one list would have 25,600 entries (at the end), so average search would appear to be equivalent to search through a list with 12,800 elements. However, the situation is worse. If the range of h has exactly 25,600 locations, then many collisions will occur (i.e., 2 distinct inputs hashing to the same address). This necessitates searching the range of h to find an unused location. To avoid a high frequency of collisions, it is desirable to allocate more than the anticipated number of distinct inputs for the range of h. The unused locations would also have to be interrogated, and the average search would be 1/2 the range. More importantly, since this secondary table method produces a contiguous list of constants, there is no wasted space after the creation process terminates. Compression cannot be applied to the resulting literal pool from direct hashing without revising the pointers already set in the endogenous matrix file.

It is to be emphasized that HEADS and LINK are no longer needed once the literal pool is constructed, so they may be returned to the "storage manager" for other usage. Further, the literal pool can be relocated to gain compactness since the stored value index is only relative to the base, which can be reset.

Summarizing, we have considered three basic elementary data structures for the endogenous matrix file, namely Basic Sequential Columnar Form (BSC), Partitioned Sequential Columnar Form (PSC), and Inverted Sequential Columnar Form (ISC). The "sequential" refers to the fact that the records are stored sequentially, and the "columnar form" means that the matrix is stored by columns (since that is to be the method of access using the primal simplex method) with the j-th record corresponding to the activity j.

2.2 GROUPING ACTIVITIES

We shall now consider a more advanced data structure designed to avoid explicit representation of certain rows by grouping activities and using their contiguity to implicitly represent certain kinds of constraints. Let suppose that we can expect some rows to be of the following form:
\[(0\ldots 0 \ 1 \ 1 \ 1 \ldots 1 \ -1 \ldots -1 \ 0 \ldots 0) \times X \Rightarrow g.\]

That is, we have a group of activities, placed in sequence, so that their coefficients in a particular row form a set of +1's followed by a set of -1's. The constraint is that this difference must relate to the "group value," \( g \), with one of the three prescribed relations \( (\leq, = \) or \( \geq) \). With scaling we can always generate one such row, but what is more important many models have a prevalence of such rows in their natural form. For example, all supply rows of a transportation problem may be put into this form, thus grouping all activities with the same source; similarly, all destination rows may be represented by grouping all activities with same destination. In these cases the set of -1's is empty, and the group value is the supply or demand values, respectively. Another type of constraint fitting this form is "balance equations," where \( g=0 \). Hence, consider the following

MARKED GROUP COLUMNAR FORM (MGC):

The General Columnar Form is used with a marker used to identify groups. The structure of a marker is similar to a header with a tag bit allocated to distinguish between a marker and header.

The marker includes information to identify the group value and the numbers of +1 members and -1 members which follows the marker.

One example of a marker is illustrated in Figure 5.

**Figure 5:** Schematic of Group Marker

<table>
<thead>
<tr>
<th>-</th>
<th>t</th>
<th>n1</th>
<th>n2</th>
<th>g</th>
</tr>
</thead>
</table>

bit 1 = tag ('-' if marker; '+' if header)
t = type of group (e.g., form of inequality)
n1 = number of +1's
n2 = number of -1's
g = group value

Let us illustrate MGC, using the above marker, with a transportation problem having 2 sources and 3 destinations. We have
Let $t = 1$ represent the group type (which is the same for each of the three groups). Note that $n_1 = 2$ and $n_2 = 0$ in each group, so MCC, with PSC for the records, produces the structure illustrated in Figure 6.

**Figure 6: Schematic of Example MG**

| $-$, $1$, $2$, $0$, $d_1$ | ... marker for group 1 (flow into $d_1$) |
| $+$, $1$, $0$, $0$ | ... header for column 1 |
| $1$ | |
| $+$, $1$, $0$, $0$ | ... header for column 4 |
| $2$ | |
| $-$, $1$, $2$, $0$, $d_2$ | ... marker for group 2 (flow into $d_2$) |
| $+$, $1$, $0$, $0$ | ... header for column 2 |
| $1$ | |
| $+$, $1$, $0$, $0$ | ... header for column 5 |
| $2$ | |
| $-$, $1$, $2$, $0$, $d_3$ | ... marker for group 3 (flow into $d_3$) |
| $+$, $1$, $0$, $0$ | ... header for column 3 |
| $1$ | |
| $+$, $1$, $0$, $0$ | ... header for column 6 |
| $2$ | |

When the groups do not overlap, such as the above example, then the structure is less complicated and certain algorithmic advantages may be deduced. This is known as "Generalized Upper Bounding." (However, even if the disjointness is violated, it is important to note the concept of using markers to represent certain row structures implicitly.) If a marker takes the same
space as three indices, then we need the groups to have more than three members to take less space than FSC without groups. Some marking schemes take less than three indices. For example, we can distinguish the case $g=0$ with one bit, thus reducing to two indices. Similarly, we can distinguish the case where $n^2=0$.

When employing a special algorithm such as Generalized Upper Bounding or Variable Upper Bounding, where disjointness of groups is assumed, the inner product algorithm need not change. The dual price associated with an equality special row is never computed since it does not effect pricing and candidate selection. (After terminating, the dual price of a special row is computed using the (modified) reduced cost of the activity.)

3. EXOGENOUS MATRIX FILES

Now let us consider the exogenous matrix file. This is designed primarily for storage and retrieval of information when the analyst is creating the problem (or revising one previously created). Therefore, we are not interested in a data structure designed for efficiency of arithmetic operations. Further, we shall be interested in access by element, by row or by column. In fact, we may be interested in access by submatrix. However, since we know that once the problem is generated, we shall convert to establish an endogenous matrix file, we may want certain information to be activity-directed. Putting the details of that issue aside for the moment, let us point out an important feature to be included.

To facilitate report writing, as well as making data base management tasks transparent, it is desirable to identify rows and columns by name, rather than merely by index number. For example, an activity corresponding to flow of goods from Dallas to New York in October may be named 'DAL.NY.OCT.' Since there may be other flows out of Dallas, one may wish to retrieve all flow out of Dallas in October by the mask, 'DAL.*.OCT' to represent a restriction on total flow out of Dallas in October.

Thus, let us take, as the basic retrieval operation, the tabulated information (i.e., array form of submatrix) from an input key of the form, (ROW MASK, COLUMN MASK). By using no asterisks (*) we are retrieving a single element. If the row mask has all asterisks, then we are retrieving the submatrix whose column names match the column mask. For the basic storage operation let us simply take the key to be (ROW NAME, COLUMN NAME). This syntax is simpler to deal with, so we shall consider it first.

A basic structure consists of a list of row names and
another list of column names, each with associated information. One way to represent this is a linear list with 3 data items per record:

ROW NAME, COLUMN NAME, VALUE.

(VALUE, here and elsewhere, can be the actual value, or it can be a pointer to a literal pool.) The difficulty with this structure is that when a new entry is declared, the entire list must be interrogated to see if it is there. This would be all right if the list remains small; however, for large problems there may be on the order of 50,000 entries, so linear search would not be very fast.

The search process can be accelerated if a sort order is maintained (say on ROW NAME first, then on COLUMN NAME). However, insertion of a new entry then becomes costly.

We shall describe a binary search tree representation which has been successful in practice. More sophisticated structures are not described here in the interest of space. Our definitions follow Knuth [6].

A binary tree is defined recursively as follows. An empty set is a binary tree. A binary tree of n nodes, symbolized by Tn, is the triple (TL, ROOT, TR), where L+R=n-1. We call TL the "left subtree" and TR the "right subtree." Associated with each node is a key whose set of values is totally ordered. A binary search tree is a binary tree whose nodes correspond to key values with the following property:

if node p is in the left subtree of node q, then KEY(p) < KEY(q);

if node p is in the right subtree of node q, the KEY(p) > KEY(q).

BASIC BINARY TREE REPRESENTATION

Each node consists of 4 fields as follows: (KEY, INFO, LLINK, RLINK), where 'KEY' is the information to be kept in order, 'INFO' is other information associated with node, 'LLINK' is the left link, pointing to the root of the left subtree, 'RLINK' is the right link, pointing to the root of the right subtree.

It is convenient to let '0' be a terminal marker, so if the left subtree of a node is empty, we set LLINK(node)=0. However, notice that the number of terminal markers in each binary tree is
precisely one more than the number of nodes. This follows from
the fact that each node, except the root, has precisely one node
pointing to it, called the "father" node, and no node points to
the root. By using a single bit (viz., the sign bit) to tag the
link field as terminal, the remaining space in the link field may
be released to provide threads back to fathers that aid in traver-
sal. We shall return to this point shortly. First, let us con-
sider basic algorithms needed to utilize a binary search tree.

ALGORITHM FINDKEY: Given an input key, K, and a nonempty binary
search tree located by ROOT, the nodes are interrogated
until the key is matched or it is discovered to be
absent. If the key is matched, then its location is
put into L and control transfers to SUCCESSFUL. If the
key is not matched, then its father node location is
put into L, and control transfers to UNSUCCESSFUL.

step 1 (Begin with root.): set L=ROOT.

step 2 (Test key.): if K > KEY(L), go to step 3;
else, if KEY(L) > K, go to step 5; else, exit
SUCCESSFUL.

step 3 (Move right.): set L1=RLINK(L).

step 4 (Test for termination.): if L1 <=0, exit
UNSUCCESSFUL; else, set L=L1 and go to step 2.

step 5 (Move left.): set L1=LLINK(L) and go to
step 4.

It should be noted that in algorithm FINDKEY we are using
the location number, rather than node number, as an index. These
can be made equal by adjusting L to account for translation and
scale (i.e., number of addressible units per record associated
with a node if record size is constant; this only requires the
INFO field to be a constant size).

Before dealing the storage and retrieval operations with the
exogenous matrix file, which uses binary search trees as its
representation, let us consider the general use of algorithm
FINDKEY. Let us suppose we wish to store a new input consisting
of (K,L), where K=KEY, and L=INFO field. If we exit SUCCESSFUL,
then this may signal an error (e.g., duplicate name which is
not allowed), or it may be the case that we wish to revise the
information by overwriting with I onto INFO field. If we exit
UNSUCCESSFUL, then this may be a new input (e.g., new column or
row declaration), and algorithm FINDKEY identifies the father node
from which we can complete the insertion (see algorithm NEWKEY.
below). Now suppose we wish to retrieve information, given K=KEY, and put this information into I. We apply algorithm FINDKEY. If the exit is SUCCESSFUL, then L locates the node, and we can extract INFO for completion of the retrieval. If the exit is UNSUCCESSFUL, then this may mean an error (e.g., input key misspelled), or we may have just wanted to check if this key is present.

ALGORITHM NEWKEY: Given (K,I) as an input key-information couple, and a binary search tree located by ROOT, (K,I) is inserted into its proper order unless it is already present, in which case control transfers to DUPLICATE (with L pointing to node). In the event no room remains to complete the insertion, control transfers to OVERFLOW.

step 1 (Setup control and interrogate tree.): Execute algorithm FINDKEY. If exit is SUCCESSFUL, go to step 2. If exit is UNSUCCESSFUL, go to step 3.

step 2 (Key found.): exit DUPLICATE.

step 3 (Location of father node found.): Fetch space from AVAIL list; if none, exit OVERFLOW; else, place location into LNEW.

step 4 (Insert.): Set LLINK(LNEW)=RLINK(LNEW)=0 and INFO(LNEW)=I. If K>KEY(L), SET RLINK(L)=LNEW; else, set LLINK(L)=LNEW.

step 5 (Set pointer.): Set L=LNEW and EXIT.

Let us now describe a binary tree representation of the exogenous matrix file. We define two such trees, one for rows and one for columns. Let ROOTR and ROOTC designate their roots, respectively. In each case the key is the name and the order relation is the lexicographical ordering by character representation (e.g. if names are letters, ordering is alphabetical). The INFO field for the row tree contains the following:

(1) row index number (based on order of entrance)
(2) row type (L,C,E or N)
(3) right-hand-side value
For the column tree the INFO field contains the following:

(1) bounds

(2) pointer to (linked) list of nonzeros.

To illustrate what we have described thus far, consider the following exogenous matrix file:

Table 3: Example Exogenous Matrix File Using Binary Tree Structures

<table>
<thead>
<tr>
<th>LOC</th>
<th>KEY</th>
<th>INFO</th>
<th>LLINK</th>
<th>RRLINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROOTR</td>
<td>S1</td>
<td>1,L,10</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>ROOTC</td>
<td>S1D1</td>
<td>11,100</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>D1</td>
<td>2,G,10</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>OBJ</td>
<td>3,N,0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>D2</td>
<td>4,G10</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>S1D2</td>
<td>12,103</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>D3</td>
<td>5,G,10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>S1D3</td>
<td>13,106</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>S2</td>
<td>6,L,15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>S2D1</td>
<td>21,109</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>S2D2</td>
<td>22,112</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>S2D3</td>
<td>23,115</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This represents the following transportation matrix:

\[
\begin{array}{ccccccc}
S & S & S & S & S & S & S \\
1 & 1 & 1 & 2 & 2 & 2 & 2 \\
D & D & D & D & D & D & D \\
1 & 2 & 3 & 1 & 2 & 3 & 3 \\
OBJ & 50 & 60 & 70 & 80 & 90 & 90 \\
S1 & 1 & 1 & 1 & \leq 10 \\
S2 & 1 & 1 & 1 & \leq 15 \\
D1 & 1 & 1 & \leq 5 \\
D2 & 1 & 1 & \leq 10 \\
D3 & 1 & 1 & \leq 10 \\
bounds: & 11 & 12 & 13 & 21 & 22 & 23
\end{array}
\]

To see how this structure works, suppose we want to interrogate the right-hand-side of row S2. Using algorithm FINDKEY, we start at ROOTR, which is location 1. Since S2>S1, we move to the right-i.e., using RLINK, we go to location 9. Now the name matches, and we can retrieve RHS(S2)=15 from the third item in INFO. Now suppose we want the nonzero list of activity S1D3. Starting at ROOTC, we discover S1D3>S1D1, so we move to the right-i.e., using RLINK, we go to location 6. Again, we discover S1D3>S1D2, so we move to location 8, where we match the column name. The second item in INFO is the pointer to the nonzero list, and its value is 106. Thus, we go to location 106, where we have a nonzero in row 1 (named S1) whose value is 1.0. The third item is the link to the location of the next nonzero for column S1D3, that is 107. This tells us row 5 (named D3) has a nonzero whose value is 1.0. The pointer is 108, so a third nonzero is in row 3 whose value is 70. Since the pointer value is 0, there are no more nonzeroes in this column.

Now let us consider the storage operation when the input is of the form, (ROW NAME, ROW INFO, COLUMN NAME, COLUMN INFO). Our task here is to update the two trees and the memory pool of nonzeroes when we receive an input in this form. If a nonzero already exists in the position identified by (ROW NAME, COLUMN NAME), or if the row has been previously declared, but with different characteristics (e.g., right-hand-side), then we must decide whether this is to be declared an error or whether a revision is intended. We shall adopt the latter.

**Algorithm Hereis:** Given (ROW NAME, TYPE, RHS, COLUMN NAME, BOUND, NONZERO VALUE),

The exogenous matrix file is updated by the implied addition or revision. If no room remains and an addition is required, control transfers to OVERFLOW.
step 1 (Find row number.): Execute algorithm NEWKEY with \( \text{ROOT} = \text{ROOTR} \) and \((K,1) = (\text{ROW NAME},* \ \text{TYPE}, \ \text{RHS})\). If exit is DUPLICATE, go to step 3; else, continue.

step 2 (New row.): Set \( M + M + 1 \) and \( \text{INFO} (L, 1) = M \), where \( L \) was returned by NEWKEY.

step 3 (Update row information.): Set \( i = \text{INFO} (L, 1) \), \( \text{INFO} (L, 2) = \text{TYPE} \) and \( \text{INFO} (L, 3) = \text{RHS} \).

step 4 (Find column.): Execute algorithm NEWKEY with \( \text{ROOT} = \text{ROOTC} \) and \((K,1) = (\text{COLUMN NAME}, \text{BOUND},*)\). If exit is DUPLICATE, go to step 5; else go to step 6.

step 5 (Look for revision): Set \( h = \text{INFO} (L, 2) \) and interrogate nonzero list (whose head is located by \( h \)). If \( i \) is present, change the nonzero value to NONZERO VALUE and EXIT.

step 6 (Add new nonzero.): add NONZERO VALUE to nonzero list (whose head is located by \( h \)), set \( \text{INFO} (L, 2) = \text{HEAD LOCATION} \) and EXIT.

The use of '*' is merely to indicate anything can be put there since its value is not part of the input, and only the system knows what its value is. For example, the row index number (step 1) and the column pointer (step 4) are internal to the data structure design. It should also be noted that the addition of a row or a new nonzero may result in OVERFLOW, already considered by algorithm NEWKEY.

To illustrate how algorithm HEREIS may be employed for matrix generation consider the transportation problem with \( M \) sources and \( N \) destinations. Let us name the \( i \)-th source row as \( S_i \) and the \( j \)-th destination row as \( D_j \); let the activity corresponding to flow from source \( i \) to destination \( j \) be named \( S_iD_j \). For example, for \( M = 2 \) and \( N = 3 \) we would have the diagram which appears on the following page.
\begin{verbatim}
S  S  S  S  S  S  
1  1  1  2  2  2  
D  D  D  D  D  D  
1  2  3  1  2  3  

OBJ x  x  x  x  x  x  
S1  1  1  1  <=x  
S2  1  1  1  <=x  
D1  1  1  1  >=x  
D2  1  1  1  >=x  
D3  1  1  1  >=x  
BOUNDS x  x  x  x  x  x  
\end{verbatim}

(\textit{where 'x' denotes an arbitrary nonzero, and 'OBJ' is the name of the objective row}).

Consider a FORTRAN-like language that uses algorithm HEREIS as a subroutine to generate a matrix, given the dimensions (M, N) and the objective values in the array, COST (which is two-dimensional with COST (i, j) corresponding to the objective value of activity SiDj). Further, let SUPPLY and DEMAND be two l-arrays to provide right-hand-sides for rows (Si) and (Dj), respectively.

\textbf{PROGRAM TRANSGEN}: This program generates the matrix for the transportation problem.

\begin{verbatim}
DIMENSION COST (10, 20), SUPPLY (10), DEMAND (20), BOUNDS 1(10, 20)

CALL INPUT (COST, BOUNDS, SUPPLY, DEMAND, M, N)

S='SQ'
DO 10 I=1, M
S=INCR(S)
D='DO'
DO 10 J=1, N
D=INCR(D)
IF (BOUNDS(I,J).LE.0)GO TO 10
A=S&D
B=BOUNDS(I,J)
\end{verbatim}
CALL HEREIS (S, 'L', SUPPLY(I), A, B, 1)
CALL HEREIS (D, 'G', DEMAND(J), A, B, 1)
CALL HEREIS ('OBJ', 'N', 0, A, B, COST(I, J))

10 CONTINUE
STOP
END

Before we look at the effects of this primitive program, note the function 'INCR' is intended to increment the name with the rules that INCR('S1')='S2', etc. The definition of the assignment 'A=S&D' is the concatenation, which we have added to the syntax.

Now consider the two trees generated. The row names appear in the order S1, D1, OBJ, D2, ..., Dn, S2, ..., Sm. This produces the following search tree:

Figure 7: Example Row Tree

The column names are entered in sort order (i.e., lexicographically increasing) to yield the following:
Figure 8: Example Column Tree

This situation is rather unfavorable if we were to retrieve on a one-at-a-time basis because the tree depth is so great.

If the DO loops are reversed in program TRANSGEN, then the order of declaration becomes:

rows: D1, S1, OBJ, S2, ..., Sm, D2, ..., Dn

columns: S1D1, S2D1, ..., SmD1, S1D2, S2D2, ...

This changes the shapes of the row and column trees and hence changes the access time. The general question of optimal trees has received some attention, but the issue here is much more specialized and is an avenue for further research.

Now let us consider basic retrieval operations in some detail. The INFO field of a node in the row or column tree may contain solution information (which is inserted upon command once the optimization process is executed). In that case retrieval is used for report writing or for initiating the optimization process using a solution obtained previously. For rows we may include the associated row activity level and the dual price; for columns we may include activity level and the reduced cost. It is generally recommended that solution information be kept in a separate file, which we can call the SOLUTION FILE, rather than included in the
exogenous matrix file. This point is not crucial to the analysis of retrieval since the structure of the solution file would essentially be the same, although the INFO field would contain different information.

If a particular row is named, algorithm FINDKEY can be used to retrieve the associated information in the row tree; the same is true for retrieval in the column tree. Similarly, if an element is to be retrieved from the couple, (ROW NAME, COLUMN NAME), then we can first use ROW NAME to retrieve the row index number; then, we can interrogate the nonzero list for the column associated with COLUMN NAME. In both portions of the retrieval, algorithm FINDKEY can be used directly. However, it is common to specify a mask rather than just one name and retrieve the submatrix (maybe just a subset of rows or a subset of columns) for any names that match. It is less efficient to retrieve one-at-a-time ignoring the fact that we have a mask and starting at the root each time, than to consider mask retrieval directly.

Let us simplify the problem by considering only a column mask as input with the output being the associated columns, say as an array. For example, consider our transportation problem just described and suppose we assign 'S1.*' as the column mask. Then, we shall retrieve the following submatrix:

```
  S   S   ...   S
  1   1   ...   1
  D   D   ...   D
  1   2   ...   n
```

We shall thread the tree with the LLINK and RLINK values at terminal markers, and we shall use the sign bit to tag the distinction between an ordinary left or right link from a (upwards) thread to a sort neighbor as follows:

LLINK(node) = sort predecessor if no left subtree.

RLINK(node) = sort successor if no right subtree.
For example, our row tree would be threaded as follows (dotted arrows for threads):

Figure 9: Example of Threaded Binary Search Tree

REPRESENTATION:

<table>
<thead>
<tr>
<th>LOC</th>
<th>KEY</th>
<th>LLINK</th>
<th>RLINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>D1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>OBJ</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>D2</td>
<td>-2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>D3</td>
<td>-4</td>
<td>-3</td>
</tr>
<tr>
<td>6</td>
<td>S2</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Note that two terminal markers are retained (viz., D1 and S2), one for the first member and one for the last member of the sort order. We can apply algorithm FINDKEY to locate the node with the smallest key value matching the mask. Then we use the threads to traverse the tree in sort order, until the node with the largest key value is visited, by applying

ALGORITHM NEXT: Given the node at location, L, the location of the node with the key in sort succession is put into LNEXT.

step 1 (Test RLINK.): Set LNEXT=RLINK(L). If LNEXT < 0, set LNEXT = -LNEXT and STOP. If LNEXT=0, exit NONE.
step 2 (Search to left.): if LLINK(LNEXT > 0, set LNEXT=LLINK (LNEXT) and repeat step 2; else, STOP.

Applying algorithm NEXT at the node with key=D3 (so L=5), we first set LNEXT=-3. Since this is negative, node 3 is determined as the successor (with key=OBJ). If we apply it to node 1, we enter step 2 with LNEXT=6. The first entrance into step 2 results in termination with LNEXT remaining equal to 6 (so S2 follows S1 in the sourt order).

The efficiency of algorithm NEXT depends upon how many times step 2 must be performed, so let us analyze this frequency.

What is the expected number of times step 2 must be performed if L is a "random" node in the tree? Knuth [6] shows that this average is 1. A sketch of that proof is as follows. First, note there are n+1 threads (or terminal links) in a tree with n nodes. Therefore, the probability that step 2 is not executed at all is (n+1)/2n, which is slightly more than 1/2. If step 2 is entered, the chances of only one execution is (n-2)/2(n-1), slightly less than 1/2. In general, the expected value is approximately

\[ E_n = \frac{1}{2} (0) + \frac{1}{4} (1) + \frac{1}{8} (2) + \cdots = \sum_{i=0}^{n-1} \left( \frac{1}{2} \right)^{i+1} \]

\[ = 1 - \frac{(n+1)}{(2^n)} = 1. \]

We have considered the important storage and retrieval operations associated with our exogenous matrix file using a binary search tree as the basic data structure. We now wish to consider additional operations associated with model revision.

Additions of rows or columns is in the mainstream of the previous descriptions, and in-place revisions (such as changing a bound value) can be handled with simple overwrite using algorithm HEREIS. What remains is a method to perform deletions.

We shall begin by ignoring threads and consider the tree in Figure 10 which appears on the following page. The keys appear inside the node, and the sort order is indicated near each node. We shall take the sort order number to be the location number during this analysis.

The easiest deletion is when the node has empty left and right subtrees, as in the case of node 6 (with key = M). Such nodes are called leaves, and they may be deleted merely by breaking the link from the father node.
Figure 10: Example Binary Search Tree

The next easiest case is when the node has one nonempty subtree, as for example node 1 (with key = B). In this case we adjust the link of the father node to point to its son, i.e., have node 3(G) point to node 2(C) instead of node 1(B), which then becomes deleted.

Now consider deleting node 5(K), which has two sons, 4(H) and 6(M). Every node in this subtree, with root at node 5, belongs in the left subtree of node 3 (the father node of the one we wish to delete). Therefore, sort order is maintained, except for the position of K, if we exchange the contents (KEY and INFO) of node 5 with one of its subtree roots. The right subtree root, node 6, may be chosen to produce:

Figure 11: Subtree for Deletion Operation
Now the node to be deleted is number 6. Since it is a leaf, we can complete the deletion.

The basic idea is to exchange the node to be deleted with one of its sons until it becomes a leaf or has only one son. In those cases deletion is trivially completed. Let us state the algorithm and then look at deleting node 3(G) in the tree illustrated in Figure 11.

**Algorithm DELETE**: Given a binary search tree and a node at location L with its father node at F such that LLINK(F) = L, this algorithm deletes the node at L while maintaining the conditions of a binary search tree.

- **Step 1**: (Test for leaf.) If node L is a leaf, set LLINK(F) = LLINK(L) and STOP; else, set L1 = LLINK(L) and L2 = RLINK(L).
- **Step 2**: (Test for empty subtree.) If L1 = 0, set LLINK(F) = L2 and STOP; else if L2 = 0, set LLINK(F) = L1 and STOP; else, go to step 3.
- **Step 3**: (Exchange.) Set T = KEY(L), KEY(L) = KEY(L1), KEY(L1) = T, INFO(L) = INFO(L1), F = L, L = L1 and go to step 1.

Let us apply algorithm DELETE to our binary search tree pictured above for L = 3(G) and F = 7(N). Since node 3 is not a leaf, we enter step 2 with L1 = 1 and L2 = 5. Both subtrees are nonempty, so we proceed to step 3 where we exchange nodes 1 and 3. This procedure leads to the following:

Figure 12: Example of Deleting a Node with Algorithm DELETE
Note that F and L were reset to 3 and 1, respectively. We next go back to Step 1. Node 1 is not a leaf, so we set L1=0 and L2=2. We enter step 2 to find only one subtree. This results in setting LLINK(3)=2, thereby deleting (logically) node 1. We then have

Figure 13: A Next Step in Applying Algorithm DELETE.

If the node to be deleted is the right son of its father (i.e., RLINK(F)=L), then a similar algorithm applies, except we exchange to the right.

In conclusion, it is to be emphasized that the binary search tree is not the only (or best) candidate for representing the exogenous matrix file. It is a useful starting point. Alternatives are beyond the scope of this tutorial.

ACKNOWLEDGEMENT

Much of the material in this tutorial is based upon unpublished works of James E. Kalan (University of Texas) and Dennis C. Rarick (Management Science Systems). I am grateful for having learned from each of them.
Bibliographic Notes

Sparsity schemes were introduced to LP codes by William Orchard-Hays (see references 4 and 9). The processing of sparse matrices has received a great deal of attention by others as well (see references 2, 11 and 12). Comparative discussions on different structures, in particular bit map versus pointer list for placement of nonzeros, has appeared in references 3, 7, 8, 9 and 10. The notion of the partitioning scheme (PSC), and the inverted file structure (ISC) is due to Kalan [5]. For broad coverage of hashing (as in ISC) and for searching (as in the operations on the exogenous matrix file) see references 2, 6 and 10. Advanced features, in particular complexity analysis, may be found in references 1 and 6.
REFERENCES


6) D. Knuth (1968), The Art of Computer Programming, Volumes 1 and 3, Addison-Wesley, Reading, MA.


12) R. A. Willoughby (1969 and 1972), Sparse Matrix Proceedings, IBM, Yorktown Heights, N.Y.