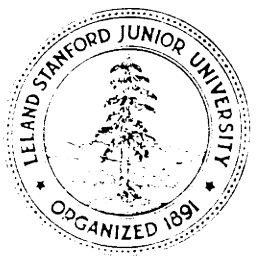
# STORING A SPARSE TABLE

by

Robert Endre Tarjan

STAN-CS-78-683
DECEMBER 1978

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



# Storing a Sparse Table

Robert Endre Tarjan*/

Computer Science Department
Stanford University
Stanford, California 94305

August, 1978

Abstract.

The problem of storing and searching large sparse tables arises in compiling and in other areas of computer science. The standard technique for storing such tables is hashing, but hashing has poor worst-case performance. We consider good worst-case methods for storing a table of $n$ entries, each an integer between $0$ and $N-1$. For dynamic tables, in which look-ups and table additions are intermixed, the use of a trie requires $O(kn)$ storage and allows $O(\log_k(N/n))$ worst-case access time, where $k$ is an arbitrary parameter. For static tables, in which the entire table is constructed before any look-ups are made, we propose a method which requires $O(n \log^{(\ell)} n)$ storage and allows $O(\ell \log_n N)$ access time, where $\ell$ is an arbitrary parameter. Choosing $\ell = \log^* n$ gives a method with $O(n)$ storage and $O((\log^* n)(\log_n N))$ access time.

CR Categories: 4.34, 3.74, 4.12, 5.25

*/ This research was supported in part by National Science Foundation grant MCS75-22870-A02 and by Office of Naval Research contract N00014-76-C-0688. Reproduction in whole or in part is permitted for any purpose of the United States government.

## 1. Introduction.

The following table searching problem arises in many areas of computer science. Given a universe of $N$ names and an initially empty table, we wish to be able to perform two operations on the table:

enter(x): Add name x (and possibly some associated information) to the table.

lookup(x): Discover whether x is present in the table, and if it is, retrieve the information associated with it.

Compilers require such a table to store names of variables [2]. Methods for LR parsing [2], sparse Gaussian elimination [6], and finding equivalent expressions [3], require such a table to store ordered pairs of integers.

In considering this problem we shall distinguish between the dynamic case, in which entries and lookups are intermixed, and the static case in which all entries occur before all lookups. We shall use a random access machine with uniform cost measure [1] as the computing model, We assume that the names are integers in the range 0 through $N-1$ and that each storage cell in the machine can store an integer of magnitude $O(N)$ .

An ideal solution to the table searching problem would be a method which requires $O(1)$ time per operation and which does not require substantially more than $O(n)$ space, where $n$ is the total number of entries made in the table. If we use an array of size $N$ to store the table, each operation requires $O(1)$ time, but the storage is excessive if $n \ll N$ . (Note that the solution to exercise 2.12 in [1] allows

us to avoid initializing the array.) If we use a balanced binary tree [4] or similar structure to store the table, the storage is $O(n)$ but each operation requires $O(\log n)$ time. The best method in many practical situations is the use of a hash table [4], which requires $O(n)$ space to store the table and achieves an $O(1)$ time bound per operation on the average, though not in the worst case.

Although for most practical purposes hashing solves the table lookup problem, it is of interest to know how far the storage required for the table can be reduced while maintaining an $O(1)$ worst-case time bound per table access. Reduction of the storage to $O(n+\sqrt{N})$, for instance, would allow storage of a $\sqrt{N} \times \sqrt{N}$ matrix with $n$ entries in $O(n+\sqrt{N})$ space with $O(1)$ access time. If the method is simple enough we may be able to beat hashing for some applications. Surprisingly little work has been done on this problem; see for example [5].

In this paper we examine two good worst-case methods of storing sparse tables. For the dynamic case, a trie data structure [4] requires $O(kn)$ storage while allowing $O(\log_k(N/n))$ access time, where $k$ is a parameter whose value is chosen in advance. The method supports table deletions as well as insertions. We discuss this method in Section 2.

In Section 3 we present a more sophisticated method which handles the static case. By precomputing the storage scheme before beginning the lookups, the method achieves an $O(n \log^{(\ell)} n)$ */ storage bound with $O(\ell \log, N)$ access time, where $\ell$ is a parameter whose value is fixed in advance. By choosing $\ell = \log^* n$ **/ we get a method

---

*/ $\log^{(1)} = \log_2 n$ ; $\log^{(i+1)} n = \log^{(i)}(\log_2 n)$ .

**/ $\log^* n = \min\{i \mid \log^{(i)} n \leq 1\}$ .

with $O(n)$ storage and $O((\log^* n)(\log_n N))$ access time. The method combines the trie structure discussed in Section 2 with repeated application of a method for compressing tables by using double displacements. This double displacement method is an elaboration of a single displacement method suggested in [2,7] for compressing parsing tables.

In Section 4 we mention some applications of our results and make a few additional remarks.

## 2. Storing a Dynamic Table.

To store a dynamic table, we use a trie [4] with n-way branching at the root and k-way branching at every other node, where $k \geq 2$ is an integer whose value is selected in advance. Each node in the trie contains one table name and either n or k pointers to nodes one level deeper in the trie. (Some or all of the pointers may be null.) Figure 1 gives an example of such a data structure.

[Figure 1]

To look up a name x in the trie, we divide x by n and then repeatedly by k. We use the successive remainders to specify a search path in the trie. For instance, to search for 190 in the trie of Figure 1, we look for 190 in the root. Not finding it, we divide 190 by 8, leaving 23 with remainder 6, which leads us to node e. Again not finding 190, we divide 23 by 4 to get 5 with remainder 3. This leads us to node i, where we find 190. To insert a name in the trie, we first search for it. The search leads to an external node, in which we place a pointer to a new node containing the new name. See Figure 1.

Our tries differ from those discussed by Knuth [4] only in that we allow the root to have a higher branching factor than the other nodes; this reduces the time required by the method without increasing the space bound, but requires that we know n (at least approximately) before we begin to construct the table. It is straightforward to implement the method, and we leave the details as an exercise. Note that by choosing the branching factors to be powers of two, we can replace division by shifting, and we can allocate space for the pointers out of a single array, avoiding initialization by using the solution to exercise 2.12 in [1].

The total space required by the method is $O(kn)$ in the worst case. The time required for either a look-up or an insertion is proportional to the length of the search path, which is $\lceil \log_k(N/n) \rceil$ in the worst case. On the average, the method requires $O(1)$ look-up and insertion time, since it is at least as fast (ignoring constant factors) as hashing with separate chaining [4].

If we add to each trie node a list of the non-null pointers in it, then our data structure will support deletions. To delete a given table entry, we first search for the node containing it, say $p$. We then locate some external node $q$ which is a descendant of $p$. We replace the entry in $p$ by the entry in $q$ and delete node $q$. If $p$ itself is an external node, we merely delete $p$. See Figure 1. With careful implementation this method requires $O(\log_k(N/n))$ time in the worst case for a deletion.

## 3. Storing a Static Table.

Section 2 shows that by using tries the worst-case time to access a table can be decreased as much as desired, at the expense of additional storage. If the table to be stored is static, i.e., all the entries take place before all the look-ups, then we can improve the method of Section 2 substantially. We shall show that for an arbitrary value of $\ell$, it is possible to store n entries selected out of N in $O(n \log^{(\ell)} n)$ space with $O(\ell \log, N)$ access time.

For simplicity we shall assume that N is a perfect square, i.e., $N = m^2$ for some integer m. We can represent the table to be accessed by an $m \times m$ array A. Position $(i,j)$ in the array corresponds to name k, where $i = \lfloor k/m \rfloor + 1$ and $j = k \bmod m + 1$. Position $(i,j)$ contains the information associated with k if k is present in the table and contains zero if k is absent from the table.

We shall describe a method for compressing A into a smaller array C, by giving a mapping from positions in A to positions in C such that no two non-zeros in A are mapped to the same position in C. Our mapping is defined by a displacement $r(i)$ for each row i; position $(i,j)$ in A is mapped into position $r(i)+j$ in C. The idea is to overlap the rows of A so that no two non-zeros end up in the same position. See Figure 2.

[Figure 2]

Each entry in C indicates the position in A (if any) mapped to that position in C, along with any associated information. To look up a name k, we compute $i = \lfloor k/m \rfloor + 1$ and $j = k \bmod m + 1$. If $C(r(i)+j)$ contains k, we retrieve the associated information. If not, we know k is not in the table. The access time with this

method is $O(1)$ ; the storage required is m for the row displacements plus space proportional to the number of positions in C . Aho and Ullman [2] and Ziegler [7] advocate this scheme as a way of compressing parsing tables, but they provide no analysis.

To use this method, we need a way to find a good set of displacements. Ziegler suggests the following "first-fit" method: Compute the row displacements for rows 1 through m one-at-a-time. Select as the row displacement $r(i)$ for row i the smallest value such that no non-zero in row i is mapped to the same position as any non-zero in a previous row. An even better method, also suggested by Ziegler, is to sort the rows in decreasing order by their number of non-zeros and then apply them first-fit. We shall employ this "first-fit decreasing" method. See Figure 2.

Theorem 1. Suppose the array A has the following "harmonic decay" property:

(H) For any $\ell$ , the number of non-zeros in rows with more than $\ell$ non-zeros is at most $n/(\ell+1)$ .

Then every row displacement $r(i)$ computed for A by the first-fit decreasing method satisfies $0 < r(i) \leq n$ .

Proof. For any row i , consider the choice of $r(i)$ . Suppose $r(i)$ contains $\ell \geq 1$ non-zeros. By (H) the number of non-zeros in previous rows is at most $n/\ell$ . Each such non-zero can block at most $\ell$ choices for $r(i)$ . Altogether at most n choices are blocked, and $0 \leq r(i) \leq n$ . □

The following algorithm is a straightforward implementation of the first-fit decreasing method. Input to the algorithm is a list of the non-zero positions in A .

First-Fit Decreasing Algorithm.

```
Step 1:  for i := 1 until m do
             count(i) := 0; list(i) := ∅ od;
         for each non-zero position do
             add one to count(i); put j in list(i) od;
Step 2:  for c := 0 until n do bucket(c) := ∅ od;
         for i := 1 until m do put i in bucket(count(i)) od;
Step 3:  for k := 0 until n+m-1 do entry(k) := false od;
         for c := n step -1 until 0 do
             for each i in bucket(c) do
                 r(i) := 0;
check overlap: for each j in list(i) do
                     if entry(r(i)+j) then
                         r(i) := r(i)+1; go to check overlap fi od;
               for each j in list(i) do
                     entry(r(i)+j) := true od od od;
```

After Step 1, list(i) is a list of the non-zero columns in row i and count(i) is a count of these non-zeros. Step 2 is a radix sort of the rows by their number of non-zeros. The initialization in Step 3 assumes that A has harmonic decay, which is the case in which we shall be interested. If A does not have harmonic decay, more space must be allocated for C .

Theorem 2. If A has harmonic decay, then the first-fit decreasing algorithm requires $O(n^2+m)$ time to compute row displacements for A ,

Proof. Steps 1 and 2 and the initialization in Step 3 require $O(n+m)$ time. For $1 \le i \le m$, let row $i$ contain $\ell_i$ non-zeros. Then the time to compute the displacement for row $i$ is $O(n\ell_i)$, and the total time to compute row displacements is $O\left(\sum_{i=1}^{m} n\ell_i + m\right) = O(n^2+m)$. □

If the array $A$ has harmonic decay, then the row displacement method provides $O(1)$-time table access while requiring only $n+2m-1$ storage, not counting storage of the information associated with each name. If $A$ does not have harmonic decay, we must smooth out the distribution of non-zeros among the rows of $A$ before computing row displacements. To accomplish this we apply to $A$ a set of column displacements $c(j)$, mapping each position $(i,j)$ into a new position $(i+c(j),j)$. This transforms $A$ into a new array $B$ with an increased number of rows (namely $\max_i c(j) + m-1$) but with the same number of columns. See Figure 3.

[Figure 3]

We choose the column displacements so as to satisfy an exponential decay condition defined as follows. Let $B_j$ be the array consisting of the first $j$ shifted columns of $A$. Let $n_j$ be the total number of non-zeros in $B_j$. Let $n_{ij}$ be the number of non-zeros in $B_j$ which appear in rows of $B_j$ containing more than $i$ non-zeros. Let $b$ be an arbitrary integer.

$$E_j(b): \text{ For } 0 \le i \le b, \ n_{ij} \le n_j/2^{i(2-n_j/n)}.$$

Note that $E_m(\lfloor \log_2 n \rfloor)$ implies $B = B_m$ has harmonic decay. To satisfy $E_j(b)$ for all $j$, we employ the first-fit method as follows:

Compute the displacements for columns 1 through m one-at-a-time. Select as the column displacement $c(j)$ for column j the smallest value such that $B_j$ satisfies $E_j(b)$. See Figure 3.

Theorem 3. The set of column displacements $c(j)$ computed by the first fit method to satisfy $E_j(b)$ for all j is such that

$0 \leq c(j) \leq 4n \log_2 b + O(n)$ for $1 \leq j \leq m$.

Proof. For any column j, consider the situation when $c(j)$ is chosen. In order for a possible choice of $c(j)$ to violate $E_j(b)$, there must be some i such that $n_{ij} > n_j/2^{i(2-n_j/n)}$. Since $E_{j-1}(b)$ holds, $n_{ij-1} \leq n_{j-1}/2^{i(2-n_{j-1}/n)}$. Each row of $B_j$ with i non-zeros in the first j-1 columns and an additional non-zero in column j contributes i+1 to $n_{ij}-n_{ij-1}$. Each row of $B_j$ with more than i non-zeros in the first j-1 columns and an additional non-zero in column j contributes 1 to $n_{i-1j}-n_{ij-1}$. Thus there must be more than

$$\left( n_j/2^{i(2-n_j/n)} - n_{j-1}/2^{i(2-n_{j-1}/n)} \right) \Big/ (i+1)$$

rows in $B_j$ with more than i-1 non-zeros in the first j-1 columns and an additional non-zero in column j. Since column j contains exactly $n_j-n_{j-1}$ non-zeros, $i > 0$. We also have

$$\left( n_j/2^{i(2-n_j/n)} - n_{j-1}/2^{i(2-n_{j-1}/n)} \right) \Big/ (i+1)$$

$$\geq \left( n_{j-1}/2^{i(2-n_{j-1}/n)} \right) \left( \frac{n_j 2^{i(n_j-n_{j-1})/n}}{n_{j-1}} - 1 \right) \Big/ (i+1)$$

$$\geq \left( n_{j-1}/2^{i(2-n_{j-1}/n)} \right) \left( 2^{i(n_j-n_{j-1})/n} - 1 \right) \Big/ (i+1)$$

$$\geq \left( n_{j-1}/2^{i(2-n_{j-1}/n)} \right) (i(n_j-n_{j-1})(\ln 2)/n) \; / \; (i+1)$$

$$\geq (i n_{j-1}(n_j-n_{j-1}) \ln 2) \Big/ \left( 2^{i(2-n_{j-1}/n)} \, n \, (i+1) \right)$$

Consider the set of ordered pairs whose first element is a row of $B_{j-1}$ with more than i-1 non-zeros and whose second element is a non-zero of column j . There are at most $n_{i-1j-1}(n_j-n_{j-1})/i$ such pairs. Each choice of c(j) for which $n_{ij} > n_j/2^{i(2-n_j/n)}$ accounts for more than $(i\, n_{j-1}(n_j-n_{j-1}) \ln 2) \Big/ \left( 2^{i(2-n_{j-1}/n)} \, n \, (i+1) \right)$ distinct pairs. Thus the number of choices of c(j) for which $n_{ij} > n_j/2^{i(2-n_j/n)}$ is bounded by

$$\frac{n_{i-1j-1}(n_j-n_{j-1}) 2^{i(2-n_{j-1}/n)} \, n \, (i+1)}{i^2 \, n_{j-1}(n_j-n_{j-1}) \ln 2}$$

$$\leq \frac{n_{j-1} \, 2^{i(2-n_{j-1}/n)} \, n \, (i+1)}{2^{(i-1)(2-n_{j-1}/n)} \, i^2 \, n_{j-1} \ln 2} \qquad \text{by } E_{j-1}(b)$$

$$\leq \frac{2^{(2-n_{j-1}/n)}\ n\ i+1}{i^2\ \ln 2} < (4\ \log_2 e)n(i+1)/i^2 .$$

Summing over i, we find that at most

$$\sum_{i=1}^{b} (4\ \log_2 e)n(i+1)/i^2 \leq (4\ \log_2 e)n(\ln b + 1 + \pi^2/6)$$

$$\leq 4n \log_2 b + O(n$$

choices of c(j) are blocked, and $0 \leq c(j) < 4n \log_2 b + O(n)$ . □

It is not hard to implement the first-fit method so that it computes column displacements to satisfy $E_j(b)$ for all j in $O(n^2+m)$ time. We leave the details to the reader.

By combining row and column displacements, we obtain the following table storage scheme.

Table Construction.

Step 1. Construct a set of column displacements c(j) for array A by using the first-fit method to satisfy $E_j(\lfloor \log_2 n \rfloor)$ for all j . Compute the transformed array B .

Step 2. Construct a set of row displacements r(i) for B by using the first-fit decreasing method. Construct the transformed array C .

Table Look-up.

Let k be the name to be accessed. Compute $i = \lfloor k/m \rfloor + 1$ , $j = k \bmod m + 1$ , and $k^* = r(i+c(j))+j$ . If $C(k^*)$ contains k , retrieve the associated information. If not, k is not in the table.

With this method, the access time is $O(1)$, the storage is $m$ for the column displacements plus $4n \log_2 \log_2 n + m + O(n)$ for the row displacements (by Theorem 3) plus $n+m-1$ for $C$ (by Theorem 1), not including space required to store the information associated with each name. The total space is thus $4n \log_2 \log_2 n + 3m + O(n)$. The time required to construct the storage scheme is $O(n^2+m)$.

If we are willing to allow a little slower access time, we can further decrease the space required to store the table, We construct not just one set of row and column displacements, but several. Each set of displacements is used to compress a different part of the table. To look up a name, we use each set of row and column displacements in turn until either finding the name or running out of mappings to try. The algorithm, described below, uses a parameter $\ell$ whose value determines the time-space trade-off.

Table Construction.

Initialization. Let $b_1 = \lceil \log^{(\ell)} n \rceil$ and for $2 \leq h \leq \ell$, let $b_h = 2^{b_{h-1}}$. Let $A_1$ be an array representing the table to be stored. For h from 1 to $\ell$, repeat the following steps.

Step 1. Construct a set of column displacements $c_h(j)$ for $A_h$ by using the first-fit method to satisfy $E_j(b_h)$ for all j. Compute the transformed array $B_h$.

Step 2. For each row i of $B_h$ containing more than $b_h$ non-zeros, let $r_h(i) = \Lambda$. Construct a set of row displacements $r_h(i)$ for the remaining rows of $B_h$ (those containing at most $b_h$ non-zeros) by using the first-fit method. Construct the transformed array $C_h$ for these rows.

Step 3. Form a new array $A_{h+1}$ by replacing with a zero each non-zero in Ah mapped to a position in $C_h$. (The non-zeros replaced are exactly those mapped into rows of $B_h$ with at most $b_h$ non-zeros.)

Table Look-up.

Let k be the name to be accessed. Compute $i = \lfloor k/m \rfloor + 1$ and $j = k \bmod m + 1$. Let h be minimum such that $r_h(i+c_h(j)) \neq \Lambda$. Compute $k^* = r_h(i+c_h(j))+j$. If $C_h(k^*)$ contains k, retrieve the associated information. Otherwise, k is not in the table.

This multiple displacement method requires $O(n^2 + \ell m)$ time to construct the table and allows $O(1)$ access time. The next theorem bounds the space required.

Theorem 4. The multiple displacement method requires $O(n \log^{(\ell+1)} n + \ell m)$ space to store the table.

Proof. For $-1 \leq h < \ell$ $b_h \geq \lceil \log^{(\ell-h+1)} n \rceil$. In particular $b_\ell \geq \lceil \log_2 n \rceil$. Furthermore, since $B_h$ satisfies $E_m(b_h)$, at most $n/2^{b_h}$ non-zeros appear in rows of $B_h$ containing more than $b_h$ non-zeros. This means $A_{h+1}$ contains at most $n/2^{b_h} = n/b_{h+1}$ non-zeros.

The storage required for the first set of displacements is m for the row displacements plus $O(n \log^{(\ell+1)} n)+m$ for the column displacements (by Theorem 3) plus $O(n)+m$ for $C_1$. For $2 \leq h \leq \ell$, the storage required for the h-th set of displacements is m for the row displacements plus $O((n/b_h) \log b_h) + m$ for the column displacements

plus $O(n/b_h) + m$ for $C_h$ . Summing over $h$ we find that the total storage is $O(n \log^{(\ell+1)} n + \ell m)$ . □

We now combine the multiple displacement method with the tree structure of Section 2 to obtain a static table storage method good for arbitrary values of n and N . Our first step is to construct a trie as in Section 2 with $k = \lfloor \sqrt{n} \rfloor - 1$ . The trie has $O(\log_n N)$ depth and contains $n + (n-1)(\lfloor \sqrt{n} \rfloor - 1) \leq n^{3/2}$ pointers, of which only $n-1$ are non-null. We can regard the pointers in this trie as consisting of a table of $n-1$ entries selected from $n^{3/2}$ possible names; $O(\log_n N)$ look-ups of pointers in this table are required to look up an entry in the original table. We use the multiple displacement scheme with $m = \lceil n^{3/4} \rceil$ to store the pointer table. We thus obtain a method which requires $O(n \log^{(\ell)} n)$ storage space and allows $O(\ell \log_n N)$ access time. If m grows only polynomially with n, the access time is $O(\ell)$ . Choosing $\ell = \log^* n$ gives an $O(n)$ -space method with $O((\log^* n)(\log_n N))$ access time.

## 4. Remarks.

There are several possible applications of our table storage schemes. The dynamic algorithm of Section 2 can be used to keep track of the fill-in when carrying out sparse Gaussian elimination [6] and to keep track of signatures when finding equivalent expressions [3]. The static algorithm of Section 3 can be used to store tables for LR parsing [2,7]. In all these applications $N = O(n^2)$. Although we have not studied the practicality of our methods, they are simple enough to be competitive with hashing in some situations. Indeed, the row displacement method described in Section 3 has been proposed as a practical way to store parsing tables [2,7]. It is important to note that our bounds are worst-case and that the worst cases are unlikely in practice.

Our algorithms make use of array storage; they cannot be implemented using only list structures as storage. Thus they indicate a difference in power between random access machines and pointer machines. They also suggest a time-space trade-off for the table storage problem, at least in the dynamic case. Whether such a time-space trade-off exists is a question deserving further study. For the static case, an affirmative answer to the following question would imply the existence of an $O(n)$ -space, $O(\log_n N)$ -access time storage scheme:

> Is there a constant $c$ such that, for any $m \times m$ array $A$ containing $n$ non-zeros, there is a set of column displacements selected from $\{0,1,2,\ldots,cn\}$ for which the transformed array $B$ has harmonic decay?

Acknowledgment.

My thanks to Yossi Shiloach for extensive discussions which contributed greatly to the ideas presented here.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[2] A. V. Aho and J. D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1977.

[3] P. J. Downey, R. Sethi, and R. E. Tarjan, "Variations on the common subeqression problem," submitted to Journal ACM.

[4] D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[5] R. Sprugnoli, "Perfect hashing functions: a single probe retrieving method for static sets," Comm. ACM 20 (1977), 841-849.

[6] R. E. Tarjan, "Graph theory and Gaussian elimination," Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York (1976), 3-22.

[7] S. F. Zeigler, "Smaller faster table driven parser," unpublished manuscript (1977).

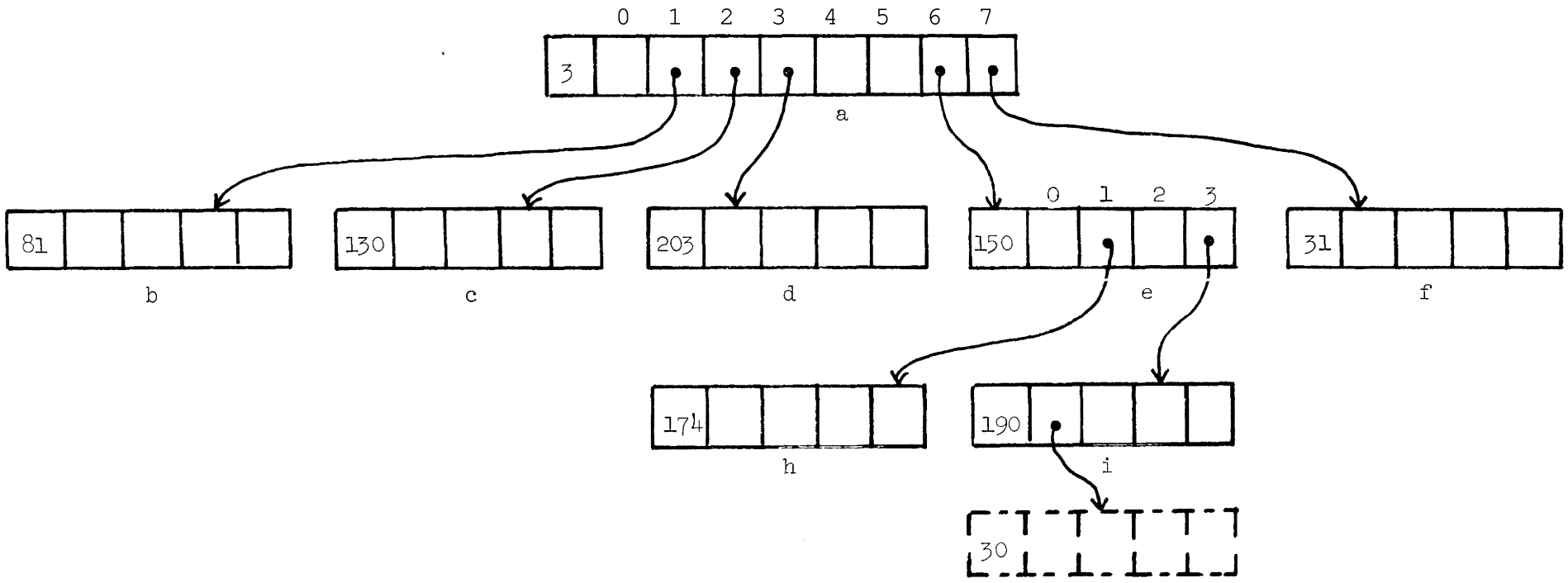



Figure 1. A trie with m = 512 , 8-way branching at the root, 4-way branching elsewhere, and entries 3 , 81 , 130 , 150 , 174, 190, 203 , 255 . To insert 30 , we divide 30 by 8 and repeatedly by 4 , giving $30 \rightarrow^{6} 3 \rightarrow^{3} 0 \rightarrow^{0} 0$ . The remainders 6 and 3 lead to node i , where we insert a new node as indicated. To delete 150 , we locate it in node e , locate an external node which is a descendant of e , say h , replace 150 in e by 174 , and delete node h .

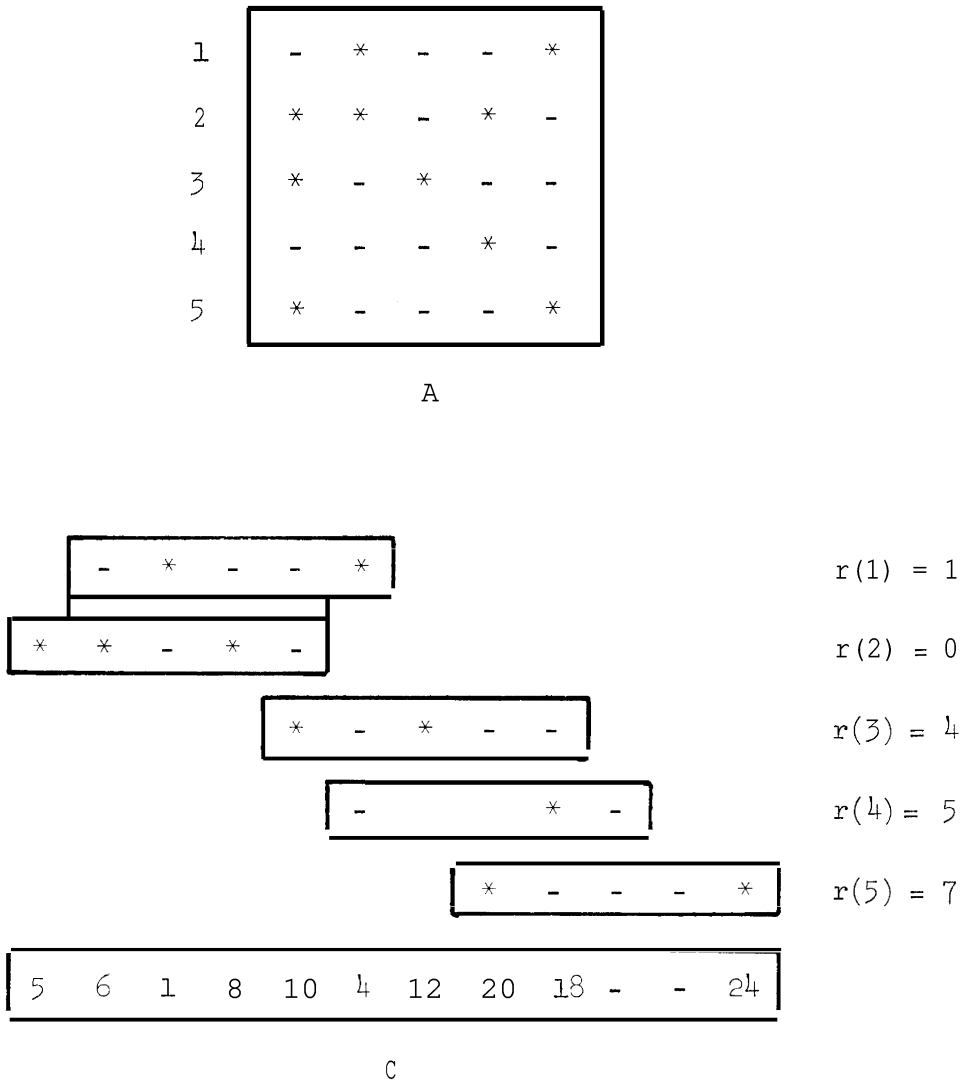



Figure 2. Row displacements computed for an array using "first-fit decreasing" strategy. Asterisks denote non-zeros; dashes denote zeros. Each position in array C contains the position in A (if any) mapped to that position in C . Positions in A are numbered row-by-row starting from zero. Row displacements are computed in the order 2 , 1 , 3 , 5 , 4 .

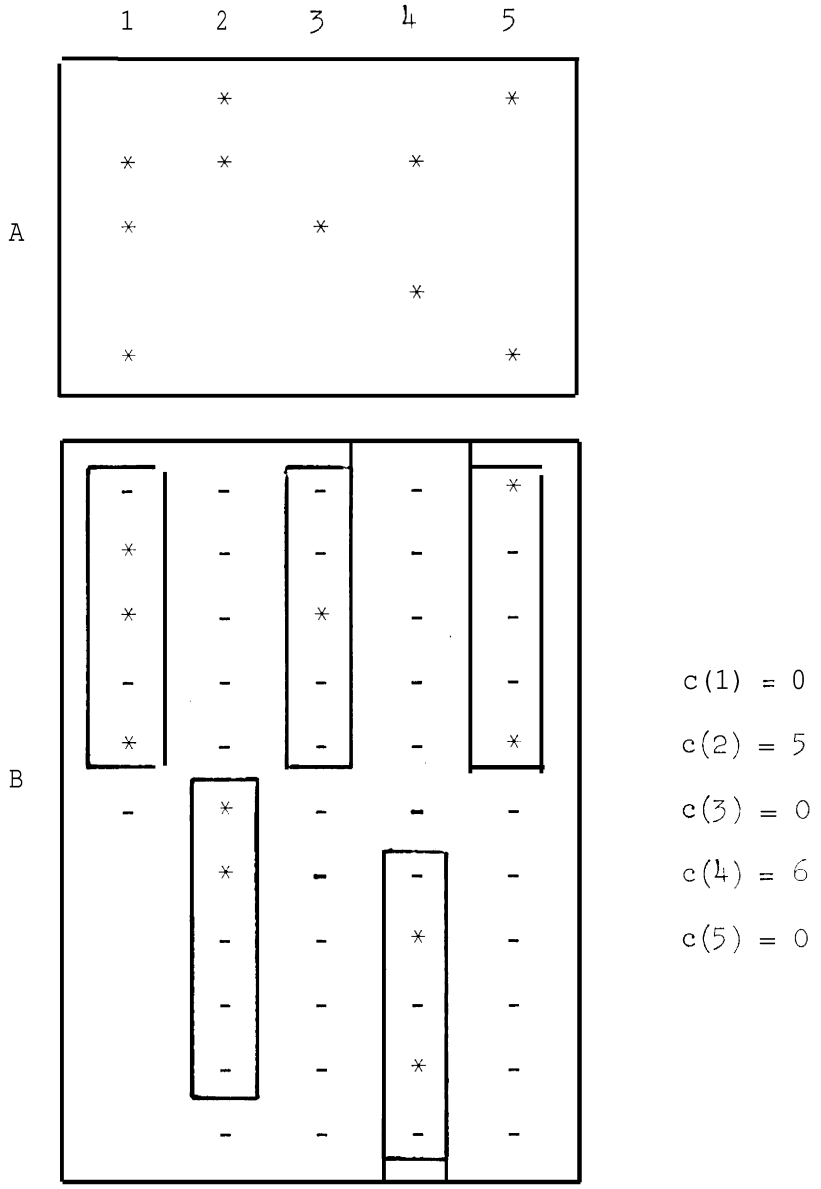



Figure 3. Column displacements computed using first-fit to satisfy $E_j(\lfloor \log_2 n \rfloor)$ for all $j$. This constraint requires no rows with more than one non-zero in $B_2$, at most one such row in $B_3$ and $B_4$, and at most two such rows in $B_5$.