BIBLIOGRAPHIC DATA SHEET	1. Report No. CU-CSD-71-111	2.	3. Recipient's Accession No.

4. Title and Subtitle

A LINEAR LIST MERGING ALGORITHM

5. Report Date November 1971

6.

7. Author(s)
J. E. Hopcroft + J. D. Ullman

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

J. E. Hopcroft
Cornell University
Ithaca, N.Y. 14850

J. D. Ullman
Princeton University
Princeton, N.J. 08540

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NSF GJ-465
NSF GJ-96

12. Sponsoring Organization Name and Address

13. Type of Report & Period Covered
Technical Rpt.

14.

15. Supplementary Notes
Also supported by: Office of Naval Rsrch Grant: N00014-67-A-0077-0021.

16. Abstracts

A linear list merging algorithm and its analysis is presented. Starting with n lists, each containing a single element, the algorithm will execute an arbitary sequence of requests to merge lists and to find the name of the list currently containing a given element. If the length of the sequence of requests is bounded by a constant times n, then the execution time of the algorithm on a random access computer is bounded by a constant times n.

17. Key Words and Document Analysis. 17a. Descriptors

merging lists

algorithm

list merging algorithm

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Distribution of this manuscript is unlimited	UNCLASSIFIED	19
	20. Security Class (This Page) UNCLASSIFIED	22. Price $3.00

Report CU-CSD-71-111

CORNELL UNIVERSITY LIBRARY
3 1924 108 114 491

A LINEAR LIST MERGING ALGORITHM

J. E. Hopcroft
Computer Science Department
Cornell University
Ithaca, New York 14850

J. D. Ullman
Computer Science Department
Princeton University
Princeton, New Jersey 08540

November 1971

Technical Report

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
Springfield, Va. 22151

A LINEAR LIST MERGING ALGORITHM*

J. E. Hopcroft
Cornell University
Ithaca, New York

J. D. Ullman
Princeton University
Princeton, New Jersey

I. Introduction

The process of merging lists occurs frequently in computer programming. The data structure used and the manner of merging can have a substantial effect on the running time of a program. Nevertheless, inefficient list merging algorithms are often used. In this paper we present an algorithm for list merging which is asymptotically faster than any previously published algorithm.

Consider the problem of representing sets by a data structure which is convenient both for merging sets and for determining which set contains a given element. A binary tree structure allows merging of sets in a fixed number of steps independent of the size of the tree but could require time proportional to the size of the set to determine which set an element is in. For purposes of comparison assume that initially there are n sets each containing exactly one item and that sets are merged in some order until all items are in one set. Then the binary tree structure has a total cost of n for the merging operations and a cost bounded by n per inquiry for determining which set contains a given element.

Methods based on maintaining balanced trees have a total cost of $n\log n$ for the merging operations and a cost bounded by $\log n$ for determining which set contains a given element. Using a linear array to indicate which set contains a given element makes

*This research was supported by the National Science Foundation Grants GJ-465 and GJ-96 and the Office of Naval Research Grant N00014-67-A-0077-0021.

the latter task finite, and by renaming the smaller of the two sets in the merging process the total cost of merging in bounded by nlogn. A more sophisticated version of the linear array replaces the set names in the array by pointers to header elements. This method, due to Stearns and Rosenkratz [3], uses

$$n \underbrace{\log \log \ldots \log}_{k}(n)$$

steps for the merging process and a fixed number of steps k independent of n for determining which set contains a given element. Here k is a parameter of the method and can be any fixed integer. The algorithm presented here performs the merging process in linear time and can answer n requests to determine which set contains a given element in time proportional to n. Although the algorithm has been used by others, this is the first time the linear nature of the algorithm has been recognized.

II. The Problem

Let us suppose we are given a set of n elements, $a_1, \ldots, a_n$. Initially, each a_i is in a set named A_i. From time to time a merge request, MERGE(A,B,C) , will be received. The merge request is executed by combining the elements of sets A and B into a single set named C. Interspersed with the merge requests are interrogation requests, FIND(a_i) which are requests to determine the name of the set of which a_i is currently a member.

This problem is an abstraction of various practically occurring computations, including the implementation of table machines [2,3]. We shall discuss applications after giving our

algorithm.

III. The List Merging Algorithm

Sets are stored in tree form, with each vertex representing one of the elements of the set. Each vertex except the root has a pointer to its immediate ancestor. The following information is attached to the root:

(1) an indication that the vertex is the root,

(2) the name of the set, and

(3) the number of members of the set.

We further assume that there are indexable arrays ELEMENT and NAME, so that ELEMENT(i) is a pointer to the vertex representing a_i and NAME(A) is a pointer to the root of the tree for set A.

To execute the instruction MERGE(A,B,C), we do the following:

(1) Find the roots of the trees for A and B, using the NAME array.

(2) Make the root of the tree with the smaller number of elements (resolve ties arbitarily) an immediate descendant of the other root.

(3) Give the remaining root the name C and set its count of members equal to the sum of the two counts.

(4) Remove the pointers in NAME(A) and NAME(B), and set NAME(C) to point to the remaining root.

To execute the instruction FIND(a_i), we do the following:

(1) Find the vertex for a_i using the array ELEMENT.

(2) Follow the path from this vertex to the root. Make each vertex along this path (except the root) an immediate descendant of the root.

(3) Print the set name found at the root.

Example: Suppose we have objects $a_1, \ldots, a_8$, and that the first four instructions are MERGE(A_{2i-1},A_{2i},A_{2i}), for i = 1,2,3,4. Then the resulting structures are as shown in Figure 1. (We have assumed that ties regarding the sizes of sets are broken in favor of the set containing the highest numbered element). Let us suppose that the instructions MERGE(A_2,A_4,A_4), MERGE(A_6,A_8,A_8) and MERGE(A_4, A_8, A_8) are executed. The result is one set, named A_8, whose structure is given in Figure 2.

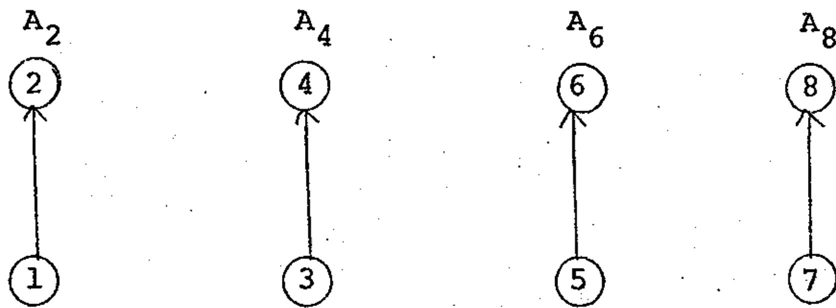


Figure 1. Tree structure after fourth merge instruction.

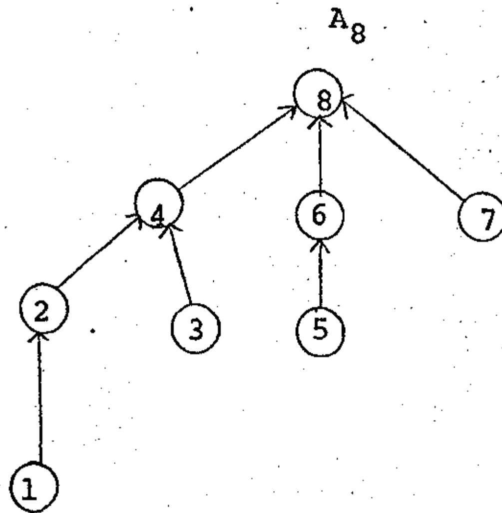


Figure 2. Tree structure after final merge instruction.

Suppose that the next instruction is FIND(a_1). Then A_8 is printed, and the vertices 1,2 and 4 are made direct descendants of the root. (The latter is already such). The resulting structure is illustrated in Figure 3.

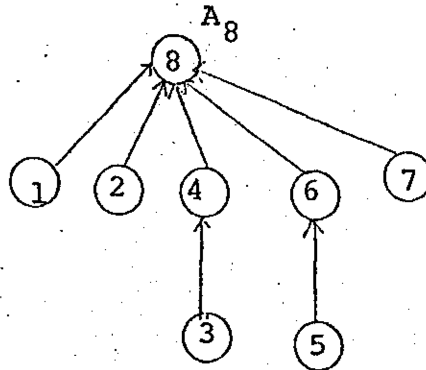


Fig. 3. Tree structure after execution of FIND(a_1).

Analysis of the List Merging Algorithm

We assume that the algorithm is implemented on a random access computer, such as [1]. It is elementary to observe that such a machine can execute any merge instruction in a constant number of steps. Moreover, the number of steps needed to execute an interrogation instruction is proportional to the number of vertices on the path from the vertex in question to the root. Since our analysis deals with order of magnitude calculations only, we will arbitrarily assign a cost of one time unit to execute a merge and ℓ time units to execute an interrogation instruction, FIND(a_i), if there are ℓ vertices on the path from the vertex for a_i to the root.

Let us now fix n, the number of elements. With this fixed value of n in mind we may make the following definitions.

Let $\alpha = x_1, \ldots, x_m$ be a sequence of instructions. Define $T(\alpha)$, the cost of executing α, to be the sum of the costs of executing each of $x_1, \ldots, x_m$ in turn. We call α normal if all interrogation instructions follow (are to the right of) all merge instructions. Let ℓ_i^α, or ℓ_i if α is understood, be the cost of executing the ith interrogation instruction in the sequence α.

Lemma 1: Let $\alpha = x_1, \ldots, x_m$ be a normal sequence of instructions. Then $T(\alpha)$, the cost of executing α, is bounded by $n+3m$.

Proof: In a normal sequence of instructions, no vertex has its immediate ancestor changed during an interrogation more than once. On the other hand, each interrogation instruction of cost ℓ moves at least $\ell-2$ vertices (all but the root and the vertex preceeding the root on the path). Thus, if i ranges from 1 up to the number of interrogation instructions, then $\sum_i(\ell_i-2)\leq n$. Noting that there cannot be no more than m interrogation instructions, the total cost of the interrogations $\sum_i \ell_i$ is bounded by $n+2m$. Since there are at most m merge instructions, the cost of all instructions does not exceed $n+3m$.

Lemma 2: Let $\alpha = x_1, \ldots, x_m$ be a normal sequence of instructions. Then $\sum \ell_i^2 \leq 15m + 13n$.

Proof: We say a vertex is of height h if the longest path from a leaf to that vertex has length h. It is a simple consequence of the fact that the merger algorithm makes the root of the smaller tree a direct descendant of the root of the larger, that any vertex of height h has at least 2^h descend-

ants (which are not necessarily immediate descendants and include the vertex itself). To see this, assume the relationship for h, then the only way for a vertex to obtain height $h+1$ is for its tree to be merged with a larger tree. Since the vertex was of height h, it must have been on a tree with at least 2^h vertices, and hence the larger tree must also have at least 2^h vertices. Thus the new tree will have at least 2^{h+1} vertices.

Form the above, we know that after the mergers in the sequence α, there can be at most $n/2^h$ vertices of height h. We observe that the square of the cost of the <u>ith</u> interrogation can be expressed as

$$\ell_i^2 = \ell_i + 2\sum_{j=0}^{\ell_{i-1}} j, \text{ since the sumation is } \ell_i(\ell_i-1)/2.$$

Thus each ℓ_i^2 is bounded above by ℓ_i plus twice the sum of the heights of the vertices on the path from the vertex interrotated to the root. Since all but the last two vertices on the path are moved by the interrogation algorithm, ℓ_i^2 is bounded above by $5\ell_i$ plus twice the sum of the heights of all vertices moved in the <u>ith</u> execution of the interrogation algorithm. (Note that the last two terms in the summation contribute

$$2((\ell_i-1) + (\ell_i-2)) < 4\ell_i).$$

Summing over all interrogations, $\sum_i \ell_i^2$ is bounded above by $5\sum_i \ell_i$ plus twice the summation over all interrogations of the sum of the heights of all vertices moved in the interrogation. Since no vertex is moved more than once during the interrogations $\sum_i \ell_i^2$ is bounded above by $5\sum_i \ell_i$ plus twice the sum over all heights h of h times the number of vertices of height h. That is,

$$\sum_{i} \ell_i^2 \le 5\sum_{i} \ell_i + 2 \sum_{h=0}^{\log_2 n} h \frac{n}{2^h} .$$

Now $\sum_{h=0}^{\log_2 n} h \frac{n}{2^h} \le 4n$, and by Lemma 1, $\sum_{i} \ell_i \le n + 3m$.

Thus $\sum_{i} \ell_i^2 \le 5(n + 3m) + 8n = 15m + 13n.$

The next lemma considers the effect of interchanging the order of a merge request and an interrogation.

Lemma 3: Let α_1 and α_2 be arbitrary sequences of instructions, and define sequences α and β by $\alpha = \alpha_1$, MERGE(A, B, C), FIND(a_i), α_2 and $\beta = \alpha_1$, FIND(a_i), MERGE(A, B, C), α_2 . Let ℓ_1 be the length of the path from a_i to the root of the tree for its set when the FIND(a_i) instruction is executed in sequence α, and let ℓ_2 be the corresponding length for sequence β. Then one of the following holds:

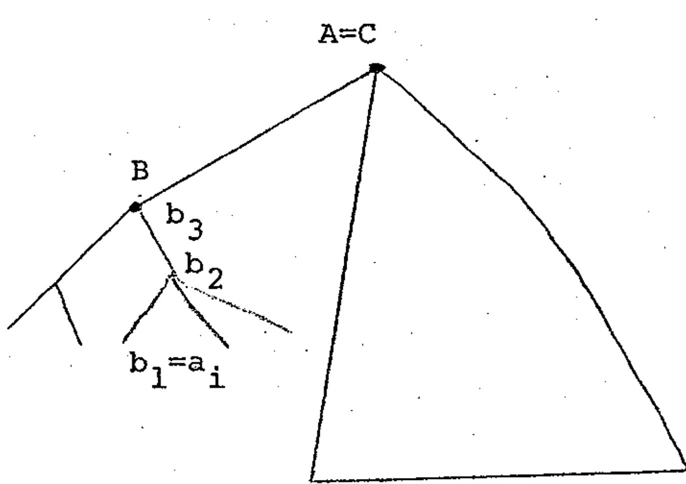
(1) $T(\alpha) = T(\beta)$ and $\ell_1 = \ell_2$, or

(2) $T(\beta) \le T(\alpha) + \ell_2 - 2$ and $\ell_2 = \ell_1 - 1$

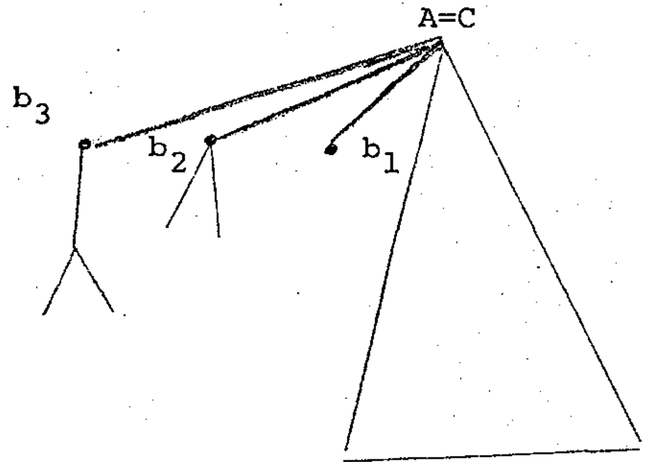
Proof: Assume without loss of generality that the set named B is smaller than the set named A. Then if a_i is not in set B immediately after executing the sequence of instructions α_1, then $\ell_1 = \ell_2$ and $T(\alpha) = T(\beta)$. On the other hand assume a_i is in B. Let the path from a_i to the root of B be $b_1, \ldots, b_{\ell_2}$, where b_1 is a_i and b_{ℓ_2} is the root of B. If MERGE(A,B,C) is executed

before FIND(a_i) then the root of A becomes the ancestor of $b_1, \ldots b_{\ell_2-1}$. If instead FIND(a_i) is exeucted before MERGE(A,B,C), then $b_1, \ldots, b_{\ell_2-2}$ are moved, so b_{ℓ_2} becomes their immediate ancestor.

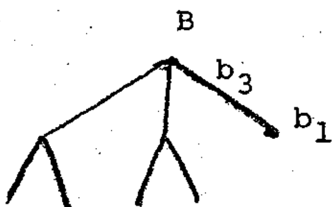
It is straightforward that in the latter case, the path from a_i includes one less vertex, the root of A, than in the former, so $\ell_2 = \ell_1-1$. This decreases the cost by one. The only other difference in $T(\alpha)$ and $T(\beta)$ stems from the fact that in β, the vertex b_{ℓ_2} may be visited ℓ_2-1 more times during interrogations occurring subsequently in β (i.e., in α_2). The first time a descendant of each b_i, $1 \leq i \leq \ell_2-1$, is interrogated in sequence β, b_{ℓ_2} will appear on the path to the root, while in sequence α, b_{ℓ_2} is skipped in each of these paths (see Figure 4). However, for any j, the second time a descendant of b_j is interrogated in α_2, b_j is the immediate descendant of the same vertex, whether the complete sequence is β or α. It thus follows that $T(\beta) \leq T(\alpha) + \ell_2-2$.

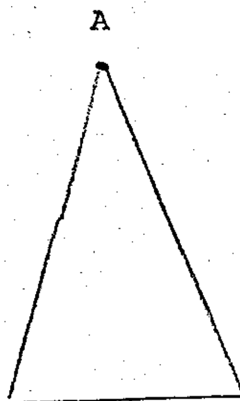


(i)

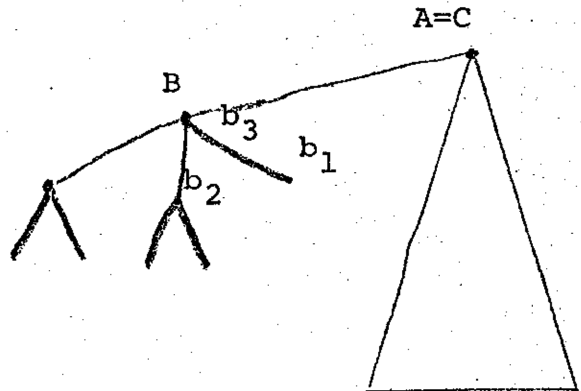


(ii)





(iii)



(iv)

Fig. 4. (i) Tree after merge in α, (ii) after FIND(a_i) in α, (iii) after FIND(a_i) in β, (iv) after MERGE in β.

<u>Theorem 1</u>: Let c be a constant, $c \geq 1$. Then there exists another constant k_c such that if β is any sequence of instructions on n objects, and the length of β does not exceed cn, then $T(\beta) \leq k_c n$.

<u>Proof</u>: Let α be the normalized sequence formed by moving all interrogations in β to the right of all merges, while preserving the order of the merge instructions and the order of the interrogations. By Lemma 1, there is a constant c_1, depending only on c, such that $T(\alpha) \leq c_1 n$.

Now let us consider what happens to the cost of the sequence, as interrogations in α are moved left over the mergers. By Lemma 3, the changes in cost incurred by moving the <u>ith</u> interrogation from its position in α to its position in β (after interrogations $1, 2, \ldots, i-1$ have been moved to their positions) does not exceed $(\ell-3) + (\ell-4) + \ldots + 1$, where ℓ is ℓ_i^{α}. Thus, $T(\beta) \leq T(\alpha) + \sum_i (\ell_i^{\alpha})^2$, where the sum is taken over all interrogations in α. By Lemma 2, there exists a constant c_2, depending only on c, such that the summation is bounded above by $c_2 n$. Choosing $k_c = c_1 + c_2$ completes the proof of the theorem.

IV <u>Applications</u>

As mentioned, Theorem 1 applies to show that table machines [3] can be implemented in time proportional to the length of the input string. The elements of the sets are the various identifiers, and the names of sets are pairs consisting of a property and a table.

Another problem to which we address ourselves is one in which merge instructions are of the form MERGE(a_i,a_j,A), meaning

"Merge the set containing a_i with the set containing a_j and call the result A." Interrogation instructions are as before. We execute MERGE(a_i,a_j,A) by following the paths from the vertices a_i and a_j to their roots, and merge the tree structures as previously. (We must be careful that we do not attempt to "merge" a set with itself). As we follow the paths from a_i and a_j, we make all vertices encountered direct descendants of the root of the tree named A.

We observe that the effect of the instruction MERGE(a_i,a_j,A) is the same as the effect of executing the instructions

MERGE(B, C, A) FIND(a_i) FIND(a_j)

by our original algorithm, where B and C are the names of the sets containing a_i and a_j, respectively, before the merge. Thus, the modified problem requires time proportional to n in the sense of Theorem 1.

In our final application we are given a sequence of instructions of the forms INSERT(i), for $1 \leq i \leq n$ and EXTRACT (MINIMUM). Consider a list which is initially empty. When an instruction INSERT(i) is executed, the integer i is added to to the list. When the instruction EXTRACT(MINIMUM) is executed, the smallest number in the list is assigned to MINIMUM and is deleted from the list. We assume the instruction INSERT(i) appears at most once in any sequence of instructions, and that at no time does the number of EXTRACT(MINIMUM) instructions executed exceed the number of insert instructions executed.

Note that as a special case, we could sort k integers from one to n by executing our insert instruction for each integer, followed by k extract instructions.

It should be observed that the algorithm which we will give is off-line, in the sense that the entire sequence of instructions must be present before processing can begin. In contrast, the list merging algorithm is on-line, in the sense that it can execute instructions without knowing the subsequent instructions with which it will be presented.

Algorithm

(1) Let $I_1, I_2, \ldots, I_k$ be the sequence of instructions to be executed. Note that $k \leq 2n$. Let ℓ be the number of insert instructions and let m be the number of EXTRACT(MINIMUM) instructions. Construct a data structure for the list merging algorithm with $m+1$ trees. The ith tree is named i and consists of the single element a_i.

(2) Construct and initialize the following arrays. A is an array of length n giving for each i, $1 \leq i \leq n$, the position, j, of the instruction INSERT(i) in the sequence. If the instruction INSERT(i) does not exist, then $A(i) = 0$. B is an array of length m giving for each i, $1 \leq i \leq m$, the location of the ith EXRACT (MINIMUM) instruction. C is an array of length k. $C(i)$ contains a pointer to the element a_{j+1} in the data structure for the list merging algorithm if exactly j of $I_1, \ldots, I_i$ are EXTRACT(MINIMUM) instructions.

(3) We will repeatedly call upon the list merging algorithm to manipulate the data structures. We do the following for $i = 1, 2, \ldots, n$ in turn.

(i) If $A(j)=0$, meaning i is never inserted, do nothing. Otherwise $C(A(i))$ contains a pointer to an element on a tree in the data structure. Execute a find instruction to obtain the name of the tree, call it t. The integer t is the extract instruction which will delete the integer i (unless $t=m+1$, in which case i is never extracted). Record the integer i in the $t\underline{th}$ entry of a table for output.

(ii) If $t \leq m$, $C(B(t))$ contains a pointer to an element on a tree whose name, s, is the first Extract instruction after I_t which does not print an integer less than or equal to i. Execute an instruction MERGE(t,s,s).

Theorem 2: The algorithm is executed in time bounded by some constant times n.

Proof: By Theorem 1, the total time in executing find and merge instructions is bounded by a constant times n. Since the remaining computation is bounded by a constant times n, the total execution time is bounded by n.

Theorem 3: The algorithm performs its task correctly.

Proof: The key point of the proof is that one may prove the following by induction on i.

After the $i\underline{th}$ iteration of step (3), $i \geq 0$, $C(j)$ points to the structure whose name is m, where the $m\underline{th}$ extract instruction is the leftmost extract instruction to the right of

I_j which results in printing an integer exceeding i.

The induction proof is elementary, and we omit details.

We note that in a variation of this problem, we could use instructions that say "print all numbers less than or equal to j in the list". We could then modify Step (3) of the algorithm as follows. After finding the tree t containing the element pointed to by $C(A(i))$, test to see if i is less than or equal to that j associated with tth print instruction. If so, skip (4ii). Otherwise perform the merge and return to step (4i) with the same value of i. Since at most n mergers are performed, the modified task can also be performed in time bounded by a constant times n.

V. Acknowledgement:

The authors would like to thank Don Knuth who pointed out the merger algorithm, with the exception of the feature that forces smaller trees to be merged into larger. That algorithm is apparently due to Bob Morris. Prior to our work the algorithm with the above feature was implemented by Doug McIlroy in a program to find sparring trees.

REFERENCES

[1] Hartmanis, J., "Computational Complexity of Random Access Stored Program Machines", Cornell University TR 70-70, August 1970.

[2] Stearns, R. E. and P. M. Lewis, "Property Grammars and Table Machines", Information and Control 14:6, pp. 524-549, 1969.

[3] Stearns, R. E. and D. J. Rosenkratz, "Table Machine Simulation," 10th Annual Symposium on Switching and Automata Theory, pp. 118-128, 1969.