Published: 01/13/67

Identification

Symbol table routines
Find_tables, lose_tables, search_tables, search_root
M. A. Padlipsky

Purpose

Find_tables and lose_tables maintain Symbol Table Lists for the debugging aids (see BD.1.00, BY.6.00). Find_tables adds a pointer to a specified Segment Symbol Table into a "master_node" structure (see below, and BY.6.00) which it keeps in static storage; lose_tables removes the pointer to a specified Segment Symbol Table from a master_node.

Search_tables is the symbol table searcher of the debugging aids. It may also be used by other routines, for searching other tables, provided the tables' tree structures are sufficiently similar to the Multics Symbol Table structure. Search_root is analogous to search_tables and is provided to allow easy handling of a "logical" table such as the Symbol Table List. The separation of the two routines is intended to allow greater generality of application, in that search_tables can be used independently of search_root for searching trees which do not have a master_node structure.

Symbol Table Lists

As mentioned in BY.6.00, the debugging aids employ the notion of a Symbol Table List to allow debugging to be performed on more than one segment (and more than one Segment Symbol Table) at a time. The actual list is kept in the branches array of the master_node structure, discussed below under Implementation. It should be pointed out that the treatment of symbol tables here is essentially one of tree-structuring.

The master_node structure may be regarded as the root of the tree while the individual Segment Symbol Tables may be regarded as sub-trees. The members of the branches array in the master_node structure are absolute pointers; the pointers in the standard symbol table entry (see BD.1.00) are relative pointers. Thus, the routines which search Segment Symbol Tables must be designed separately from those which search the Symbol Table List, in order to deal with the structure encountered.

.Find_tables, Lose_tables

Usage

The calling sequences are

```
call find_tables (name, node_pointer);
call lose_tables (name, node_pointer);
```

with declarations as follows:

```
dcl name char (*) varying, node_pointer ptr;
```

Name is the name of the segment whose symbol table is to be "added" to (or "deleted" from) the Symbol Table List. What actually happens is that a pointer to the segment symbol table is placed in (or removed from) the master_node structure which represents the Symbol Table List.

Node_pointer is a pointer to the master_node structure in use:

a) On the initial call to find_tables (which is from probe, in the context of the debugging aids), the calling program must set node_pointer to "null"; in this case, find_tables immediately allocates a master_node and subsequently returns with node_pointer pointing to it. On later invocations of find_tables, node_pointer is also returned; the calling program must be aware of this fact, as possible reallocation of larger master_node's can alter the value of node_pointer. Probe, for example, will use node_pointer as the starting node to be communicated to arg when a search of an entire Symbol Table List is desired (see below, regarding search_tables).

b) On calls to lose_tables, node_pointer is strictly an input argument, and is, of course, to be set to (or retained at) the value received from the last call to find_tables.

Implementation

Figures 1 and 2 present block diagrams of find_tables and lose_tables.

"Master_node" is declared as follows:

```
dcl  1 master_node ctl(p), 2 n fixed, 2 m fixed,
     2 branches (p->master_node.n)ptr, 2 info_pointer
       ptr, 2 back_pointer ptr;
```

where n is the length of the branches array, m is the number of branches currently in use, branches is an array of pointers to Segment Symbol Tables, and info_pointer and back_pointer are not used by the debugging aids, but are provided for possible generality and for compatibility with the symbol table nodes.

The logic of find_tables is as follows: if the node_pointer passed to the routine is null, a master_node structure must be allocated; six slots are allowed for the branches array, and the number of slots used is set to zero; node_pointer is set to point to the structure. If node_pointer is originally non-null, a check must be performed to determine whether the branches array is full; if so, a new master node is allocated with an array six slots longer, and the old information is set into the new structure (except for the array length, of course); the old master_node is freed, and node_pointer is set to point at the new master_node. After questions of allocation have been dealt with, a pointer to name's symbol table is generated by calls to the Multics equivalents of linkmk and link (see BE.8.01, BE.8.04), the number of slots used is updated and the pointer is set into the branches array in the first unused slot. (If the pointer cannot be generated because of inability to find name's symbol table, condition "find_tables_001" is signalled; a return is provided in the event that the signal returns.) The decision to allocate the branches array six slots at a time is dictated by the inefficiency of allocating; the choice of six is arbitrary.

Lose_tables is rather more straightforward: the symbol table pointed at by each member of the branches array is investigated until either name's symbol table is found or all members of the array have been dealt with. If the desired symbol table is found, the remaining members of the branches array are rewritten into the next-lower-numbered slots, the number of slots used is set to one less, and the routine returns. If name's symbol table is not found, condition "lose_tables_001" is signalled; a return is provided, in the event that the signal returns.

Search_tables, Search_root

General

The basic task of search_tables and search_root is the

locating of a specified symbol in the tables comprising a Symbol Table List. The intended caller for both routines is the arg subroutine of evaluate (BY.6.04). If the search is to cover the entire Symbol Table List, search_root is invoked; it, in turn, invokes search_tables. If the search is to cover a particular segment or "block" (in the PL/I sense), search_tables is invoked directly. The calling sequences are

```
call search_root (start_node, name, verify, no_name);
call search_tables (start_node, name, verify, no_name);
```

with declarations as follows:

```
dcl start_node ptr, name character (*) varying, verify
    entry(ptr,ptr), no_name bit (1);
```

Start_node is a pointer to a node in a Symbol Table List or in the tables comprising the List. In the initial invocation of search_root, the node at hand is the master_node of a Symbol Table List. On the initial invocation of search_tables, the node at hand may be the root node of a particular segment or block. In the debugging aids, the starting node and a switch indicating whether it is a master node are communicated to arg via static storage; arg then passes the desired starting node to search_root or search_tables (as appropriate) as the argument start_node. The static variable may, for instance, have been set by probe (BX.6.04), in response to the appearance in an expression of the "?" operator of the debugging language (BY.6.03, BX.10.00).

Name is the symbol to be searched for. It is set by arg from the "pole"-structure at hand (see BY.6.03, BY.6.04).

Verify is an entry (probably an internal procedure in the calling procedure) where a "find" in search_tables is checked for acceptability; e.g., the information block is interpreted and if a variable is desired but a "block" (in the PL/I sense) has been found, the caller's verify routine does a return, which indicates that search_tables is continue searching. Otherwise, the verify routine transfers to the appropriate location elsewhere in the calling procedure. ("Appropriate" is meant to imply either the calling procedure's return to its caller or the calling procedure's further processing of the found symbol. Before return, arg, for example, must get the value associated

with the symbol, once the symbol has been found by search_tables.) It should be noted that the mechanism of the verify entry in the calling program is necessary in order to relieve search_tables of the onerous (and unmodular) chore of interpreting translator-dependent information blocks. The calling sequence for verify is

```
call verify (node_pointer, info_pointer);
```

with declarations as follows:

```
dcl (node_pointer, info_pointer)ptr;
```

Node_pointer is a pointer to the symbol table node whose associated information block contains a name which matches the name argument; info_pointer is a pointer to the information block itself. Although it can be found on the basis of node_pointer, info_pointer is passed to verify in order to save some execution time - as it is available to search_tables already.

No_name is a switch included for the convenience of calling programs which want to look at the symbol table per se (caterpillarize the tree, perhaps?); if it is set to "1"b, search_tables does not compare on name but traces through the tree structure from the original start node, calling verify and furnishing new node_ and info_pointers at each recursion; if it is set to "0"b, search_tables proceeds normally, comparing on name and only calling verify when a match is found. (Arg always calls search_tables with no_name set to "0"b.) In the no_name equal to "1"b case, search_tables returns to its caller when the remainder of the tree (from start_node) has been traced through (unless verify takes other action, of course). Such a return where no_name was originally equal to "0"b, however, indicates the "not found" condition, and the calling routine must take appropriate action.

Strategy

The search is carried out as straightforwardly as possible. Given a start_node, the name in its associated information block (see BD.1.00) is compared with the name being searched for. If they are the same, the search is done. Otherwise, search_tables looks at the next inferior node (inferior to start_node, that is), the next inferior node to that, and so on until the "limb" of the tree is exhausted. After exhausting a "limb", search_tables investigates any offshoot limbs in a similar superior-to-inferior fashion. This approach may seem rather feudal, but it lends itself

quite handily to recursive implementation. Figure 3 offers an example of the search strategy. The numbers indicate order of search, solid arrows indicate pointers, circles are information blocks, rectangles containing "X" 's are "links", in the sense of section BD.1.00, lettered boxes indicate nodes, and broken arrows indicate back pointers (for convenience, they are shown in the case of terminal nodes only).

Implementation of search_tables

Figure 4 presents a block diagram of search_tables. Two comments are perhaps in order. In the first place, the decision not to search links is subject to change. For while the evaluate-oriented application of search_tables must not search links (by definition, "branches" contain the information of interest to the debugger; see BD.1.00) it is possible that some non-debugging aids user of the "no_name" mode of searching might find a different policy useful. At present, however, this does not appear to be the case, and links are rejected out of hand. Second, the "node.n" and "node.branches(i)" notation assumes that the entries in the symbol table (see BD.1.00) can be treated in PL/I by judicious use of declarations and unspec's, or by a (currently proposed) built-in ability to handle relative pointers directly in PL/I. Should this not prove to be the case, a machine language subroutine will be necessary to map the entry into a "node"-structure, with "derelativized" pointers.

The logic is as follows: Check the no_name switch: if it is on ("1"b), set node_pointer equal to start_node, set info_pointer equal to the information block pointer of the node at hand, and call the verify entry. (A transfer around the comparison on name must be provided after the call to verify, so that verify can return to search_tables and allow the searching to continue.) If no_name is off ("0"b), compare the name argument with the name entry in the information block associated with the node pointed to by start_node. If they are the same, proceed as in the no_name on case; note, however, that verify will return only if the verification is not satisfactory. Otherwise, check to see if the node at hand has any branches and/or "links" (start_node node.n>0); if not, return to caller; if so, set up a "DO" loop for start_node node.n iterations and proceed to call search_tables recursively with each of the branches(i) as start_node. (The call is not made, of course, if the branches (i) is a "link".) When (actually, "if") the loop finishes, return to caller. The recursion

is set up such that if any of the returns from search_tables (or from search_tables via search_root) turns out to be the original caller instead of to a "dynamic ancestor" (see BY.6.04), the search has been completed and a "not-found" condition exists.

Implementation of search root

Figure 5 presents a block diagram of search_root. It is very straightforward. (Master_node is described above, and in section BY.6.00). The issues of "derelativizing pointers" and reading information blocks do not arise at this level, as master_node is a PL/I structure with "real" pointers and an information block that is irrelevant to the debugging aids.

Figure 1.

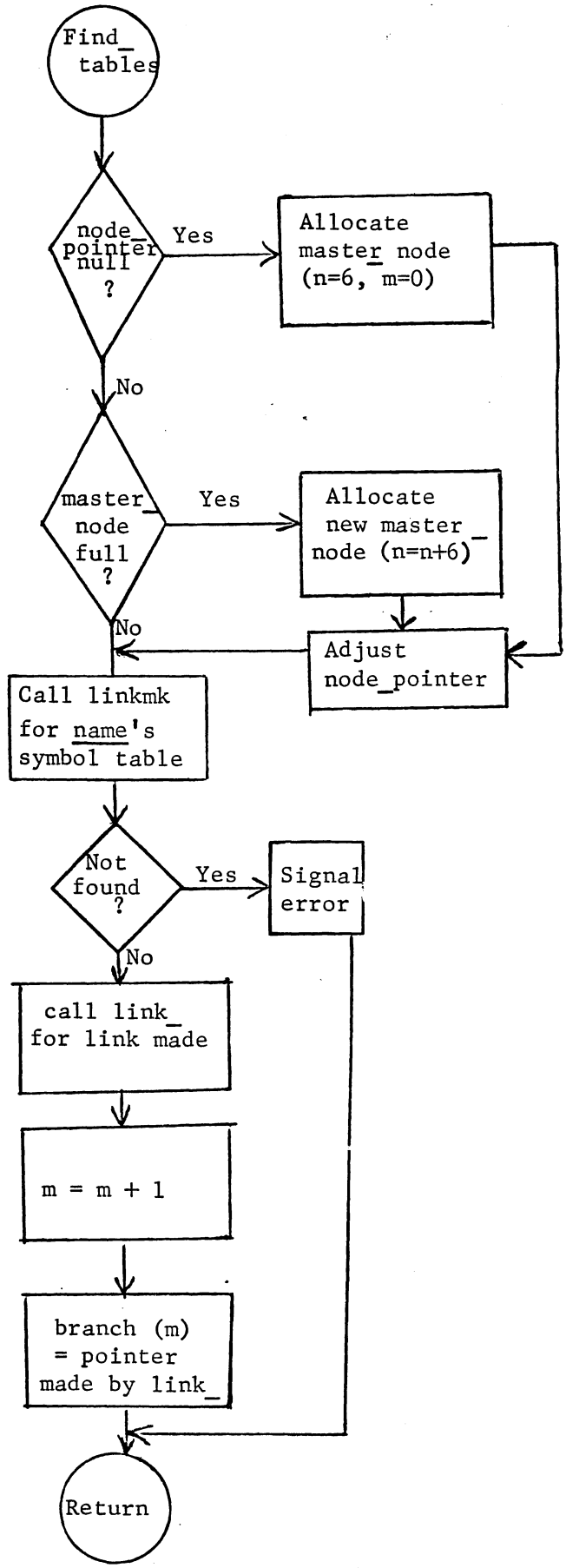

Find_ tables
node_ pointer null ?
Yes
Allocate master_ node (n=6, m=0)
No
master_ node full ?
Yes
Allocate new master_ node (n=n+6)
No
Adjust node_pointer
Call linkmk for name's symbol table
Not found ?
Yes
Signal error
No
call link_ for link made
m = m + 1
branch (m) = pointer made by link_
Return

Figure 2

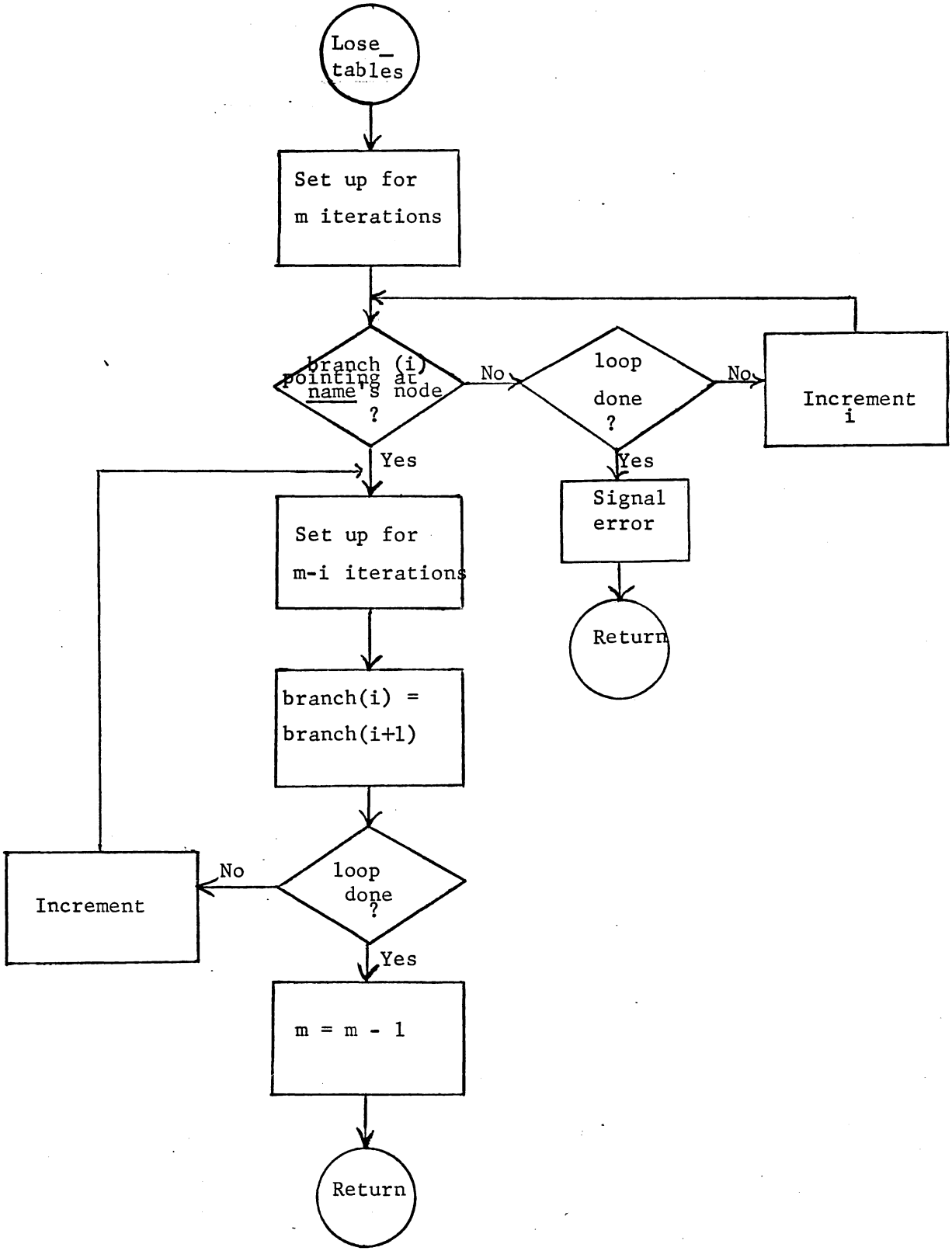

Lose_ tables
Set up for m iterations
branch (i) pointing at name's node ?
No
loop done ?
No
Increment i
Yes
Yes
Signal error
Return
Set up for m-i iterations
branch(i) = branch(i+1)
Increment
No
loop done ?
Yes
m = m - 1
Return

Figure 3 Search Order

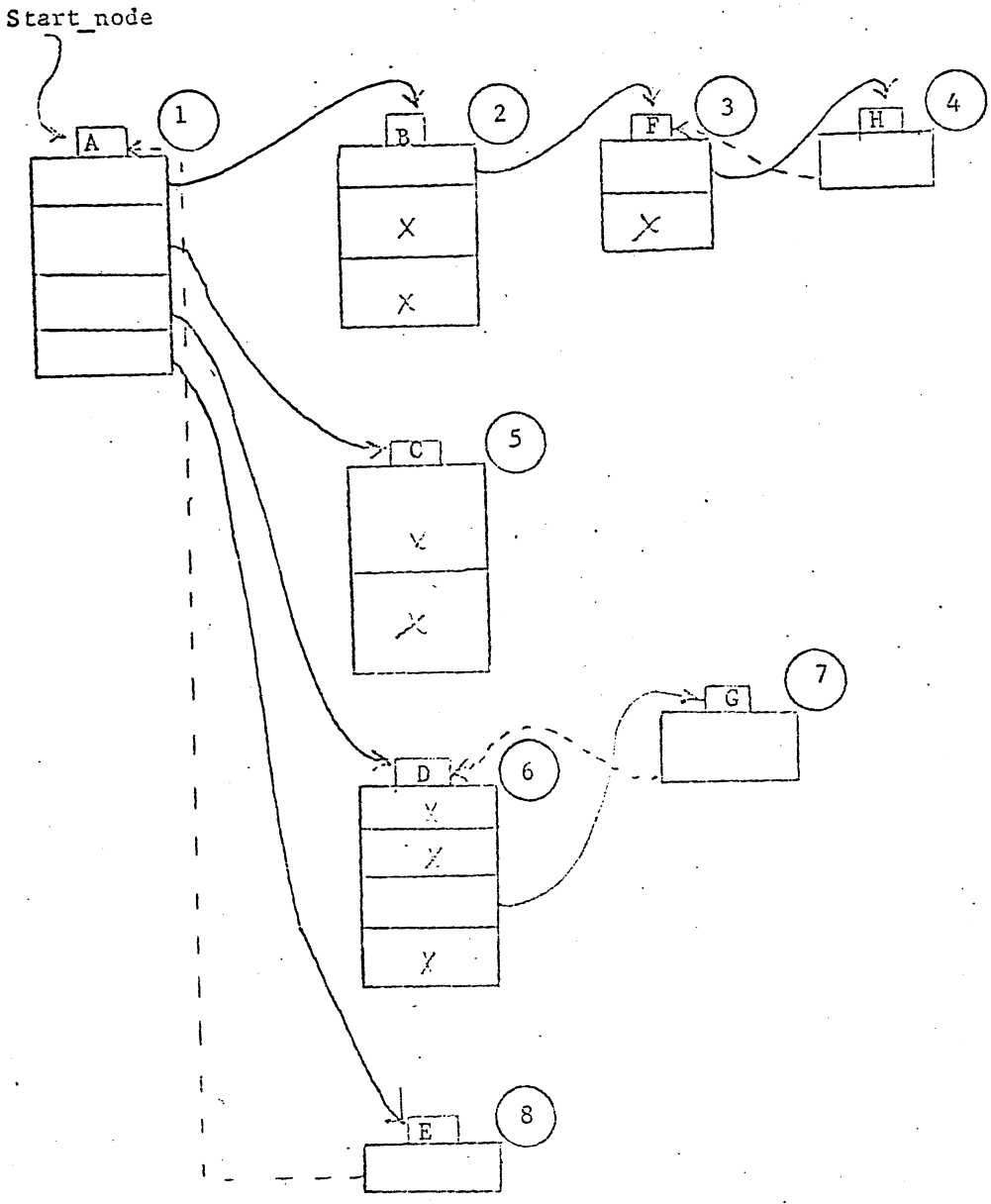


Figure 4

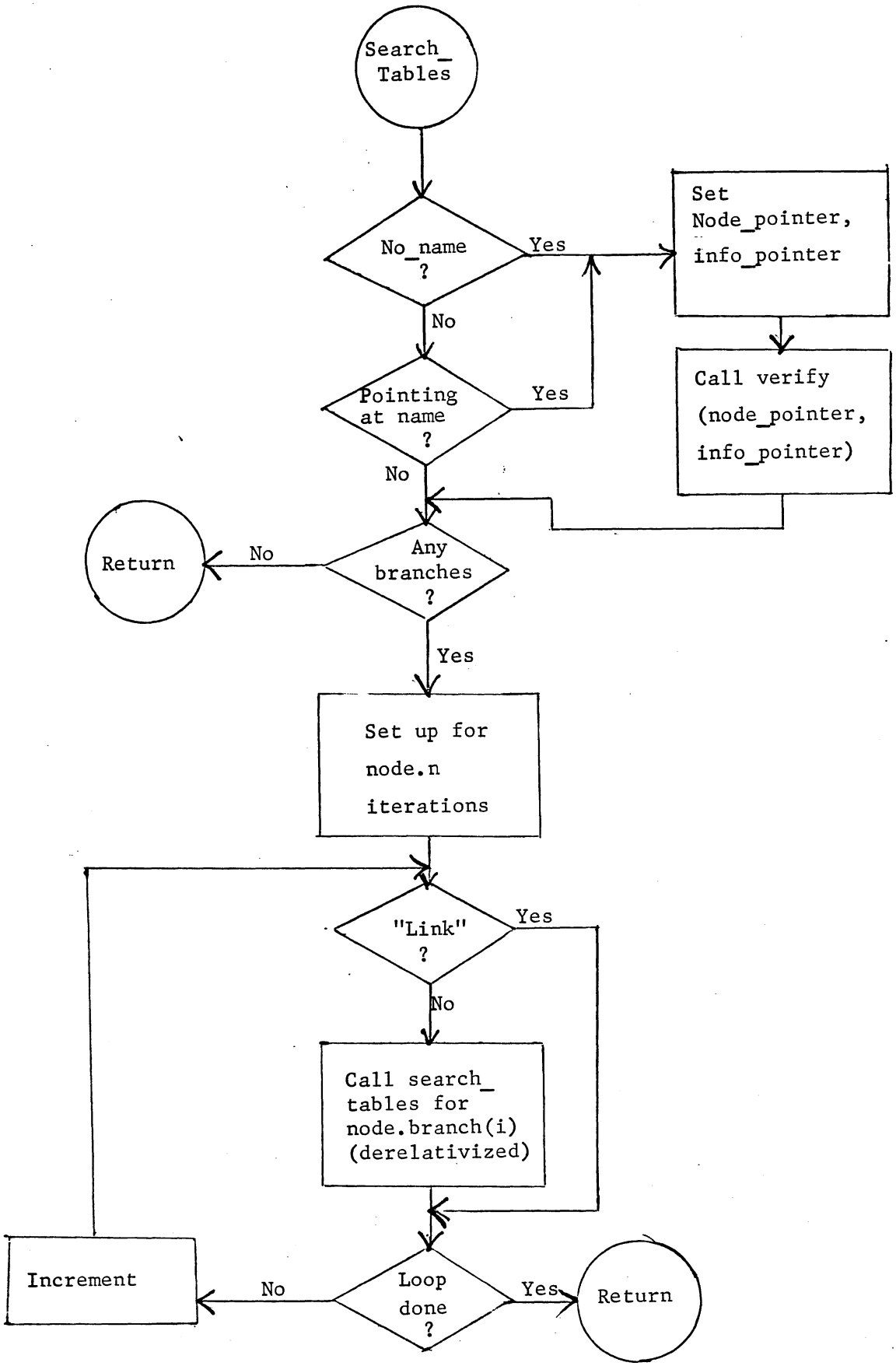
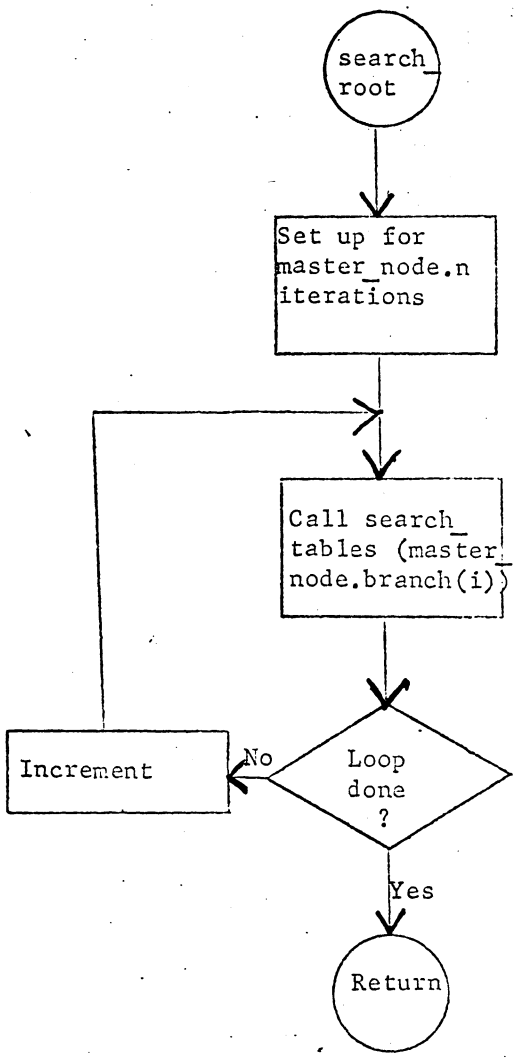

Search_ Tables
No_name ?
Yes
Set Node_pointer, info_pointer
No
Pointing at name ?
Yes
Call verify (node_pointer, info_pointer)
No
Return
No
Any branches ?
Yes
Set up for node.n iterations
"Link" ?
Yes
No
Call search_ tables for node.branch(i) (derelativized)
Increment
No
Loop done ?
Yes
Return

Figure 5



(Indicates symbol not found)