PUC

Series: Monographs in Computer Science
and Computer Applications

Nº 5/70

BASIC FUNCTIONS FOR HANDLING BINARY TREES

by

Arndt von Staa

Computer Science Department - Rio Datacenter

CENTRO TÉCNICO CIENTÍFICO
Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

ABSTRACT

In this paper several basic subroutines for handling binary tree structures are presented. The subroutines may handle threated binary tree representations of trees and/or normal binary tree representations. The described functions perform the following activities: pointer handling; insertion and creation of nodes; copy and deletion of trees; conversion from normal tree to threaded tree representation; traversing of threaded binary trees; schematic printing of trees. Some utility subroutines are included.

The main algorithms are presentad and the use of the subroutines is described.

INDEX

1 - PURPOSES:

This set of subprograms was designed to aid the programmer in the use of structures such as binary trees. Several of these subprograms may be used as a special aid in conventional programming.

The set of subroutines contains:

a - subroutines for handling the storage area.
b - subroutines for handling pointers.
c - subroutines for the creation, copy or deletion of nodes.
d - subroutines which transform a normal binary tree into a threaded binary tree.
e - subroutines for traversing threaded binary trees.
f - subroutines to obtain a schematic printing of a binary tree.
g - utility subroutines.

2 - DEFINITIONS:

a - Node: is a complete description of a single element of the binary tree and consists of 4 fields: Information, left link, right link, free tag.

b - Null pointer: is a pointer of a node which has no offspring at the side where the null pointer is located. If the left

subtree of a node is empty, the left link field contains a null pointer. If the right subtree of a node is empty, the right link field contains a null pointer.

In this set of subprograms, any pointer value less then one will be a null pointer.

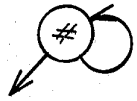
c - Left terminal: is any node for which the left link pointer is a null pointer. This rule also applies to right terminal.

d - Terminal node: is a node in which both link fields contain null pointers.

e - Head: is a special node in which the information field contains a special flag. This flag is any integer value greater than or equal to $377\ 777\ 777\ 770_8$ or $34\ 359\ 738\ 360_{10}$. Any node containing such a value will be treated as a head. Head nodes are used with threaded binary trees.

A head node points to the left to designate the tree of which it is head. A head node points to the right to itself. If the tree of which it is head is empty,

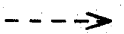
the left pointer will be negative and its absolute value will point to the head.



Head of a non empty tree

Head of an empty tree

pointer, positive valued pointer



thread, line, negative valued pointer

Figure 1

f - Threaded binary tree: is a binary tree of which the terminal links, token in absolute value, point to the predecessor or sucessor in infix traversing order. The left terminal will point to the predecessor in infix order, the right terminal will point to the successor in infix order.

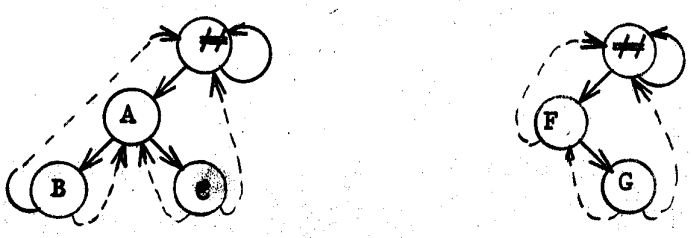


Figure 2

g - Thread line: is a null pointer of a threaded binary tree. Its absolute value points either to the predecessor, if in the left field, or to the successor, if in the right field,of a node. A thread line will always be negative.

3 - WORKING STORAGE:

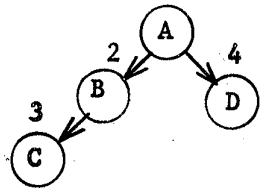
The working storage contains all the nodes of all the defined binary trees, and also a free space stack. The working storage is defined in a CØMMØN area. This CØMMØN area normally contains space for 100 nodes. If the CØMMØN area is too small, the user may alter the space without changing any of the subprograms. This is done by defining in the main program a CØMMØN area with dimensions different to those normally used. For example, the declaration:

```
COMMON /BTREE/ IARV(2,NN)
```

where NN is the ammount of nodes to be used and IARV is the working storage.

Each node occupies two words of the IBM-7044. The elements of IARV with line index equal to 1, contain the information of the node. The elements of IARV with line index equal to 2 (IARV(2,XX)) contain both link fields (pointers) and the free-space tag of the node. A node is pointed by its

column index.



	1	2	3	4
IARV(1,X)	A	B	C	D
IARV(2,X)	2	3	0	0
	4	0	0	0

Figure 3

Nodes should be obtained from or returned to the free space stack only by means of the corresponding subroutines. The subroutines will check the validity of the nodes returned to or obtained from the free space stack.

3.1 - Subroutines for Handling the Free Storage Space*:

a - Obtaining a node from the free space.

The subroutine DLIVRE(IP) returns, in IP, a pointer (column index) to a free node. The subroutine checks to find it the node is really free, or if it is erroneously in the free space stack. The fields of the node pointed by IP will be set to zero.

*The subprograms were designed to run under IBSYS of the IBM-7044, version 9 modification 19.

The free space need not be initialized, even if the size is not the normal one, since the check for storage overflow is automatically adjusted.* But if adjacent CØMMØN areas are used the overflow test may fail.

b - Returning a node to the free space.

The subroutine PLIVRE(IP) returns the node pointed by IP to the free space stack. If the node IP already belongs to the free space stack, a message will be printed and the subroutine will return immediately to the calling procedure.

c - Initialization of the free space.

The subroutine ESPACØ(NN) initializes a free space which will contain exactly NN nodes. The value of NN should be equal to the number of nodes previewed by the CØMMØN /BTREE/IARV(2,NN).

The use of the subroutine ESPACØ is optional. It should be used when there is uncertainty of well functioning of DLIVRE. The subroutine ESPACØ inhibits the creation of free space by the DLIVRE subroutine, therefore the use

---

*During a normal processing of a job under the IBJØB language processor, all areas are set to zero during the load phase.

of ESPACØ avoids errors in detection of space overflow.

The subroutine ESPACØ may be used to reset the whole working storage. A call to ESPACØ will reset and set available NN nodes. It can be seen that all information in the working storage area is lost when ESPACØ is called. When ESPACØ is called at least one node must remain available, or a space overflow message will be printed.

d - Final Remarks:

As no garbage collection is made, the responsibility for the use and return of a node, is left to the user.

4 - LINK FIELDS:

4.1 - General Aspects:

The pointers are indexes and always integer. They give the column index of the pointed node. Both pointers of a node, left and right, are stored in one word. This word always has its line index equal to 2.

The value of a pointer depends on the size of the working storage. The range of values which a pointer may hold, varies from -16383

to + 16383 including zero.

The aspect of the word which contains the links is as follows:

Bit	0	1	2	3	4 ... 17	18	19	20	21	22 ... 35
	U	MK	ID	LS	LEFT	U	U	U	RS	RIGHT

Where:

U - are unused bits

ID - is a tag which indicates wether a node is free (ID=1) or used (ID=0)

MK - is a bit used by the tagging algorithm

LS - is the sign of the left pointer

RS - is the sign of the right pointer

LEFT - is the absolute value of the left pointer. LS and LEFT perform the left link field

RIGHT- is the absolute value of the right pointer. RS and RIGHT perform the right link field

Pointers should be altered or consulted only by means of the pointer handling subroutines.

4.2 - Subroutines to Set New Values into the Link Fields:

The subroutines ESQLIG(IP,IQ) insert a new value IQ into the left (ESQLIG) or the right (DIRLIG) link field of the node pointed by IP.

4.3 - Subroutines to Obtain the Value of a Link Field:

The functions IESQ(IP) and IDIR(IP) return the value of the left (IESQ)or the right (IDIR) link field of the node pointed by IP. The returned value of both functions is always an in - teger. The node IP remains unchanged.

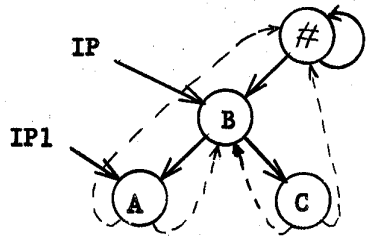
5 - CREATION, INSERTION OR COPY OF NODES OR BINARY TREES:

5.1 - Insertion of Nodes into Threaded Binary Trees:

The subroutines ESQATA(IP,IQ) and DIRATA(IP,IQ) insert a node IQ on the left (ESQATA) or right (DIRATA) side of the node IP. The node IP has to belong to a threaded binary tree. The node IP may be anyone of the nodes of a threaded binary tree, except when the subroutine used is DIRATA and in this case the node IP cannot be a head node. The node IQ must not a be null pointer or a head node. IQ should only be a single node. If IQ is a root of a subtree, the structure of this subtree

will be destroyed. After using ESQATA or DIRATA the resulting tree will be a threaded binary tree, even if IP is not a terminal node.

Given the threaded binary tree:



→ pointers, positive valued

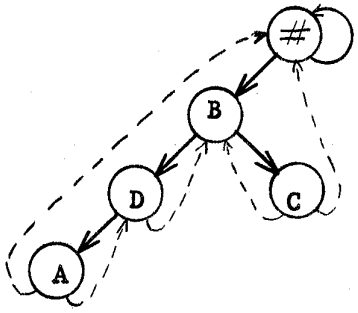
- - → thread lines, negative valued

IP and IP1 are pointers to nodes of the tree.

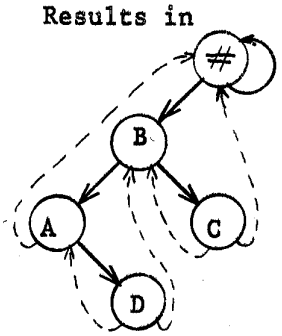
And given the single node IQ → D

The statements:

CALL ESQATA (IP, IQ)
Results in



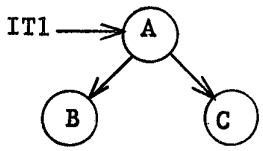
CALL DIRATA (IP1, IQ)
Results in

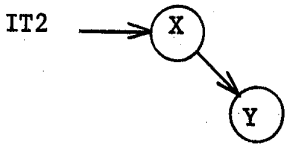


5.2 - Construction of a new Root:

The function ICNS(INFØ, IL, IR) constructs a new root of the subtrees with roots point by IL, at the left, and IR at the right. The information field of the new node contains the value INFØ after construction. The value returned by this function is the pointer to the new root. The function ICNS is not used for threaded binary trees. The subtrees IL and IR forwhich ICNS will construct a new root may be empty,in this case they should be disjoint. There is no check to find if IL or IR point to a head node.

Given the trees

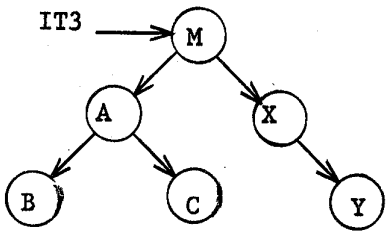




The statement:

IT3 = ICNS(M, IT1, IT2)

Results in:



5.3 - Copying of Trees or Subtrees:

The subroutine CØPIA(IP, IQ, ISIDE) copies a binary tree or subtree with root IQ. The copy may produce a new tree identical to that with root IQ. The copy may also add at the right side or at the left side of a node IP a complete new subtree identical to that with root IQ. If the copy is to be complete, the argument ISIDE shall contain the character I in format A1(1HI). In this case IP may contain any value, IP will be set properly to point to the root of the new tree.

If the copy is to be added either at the left or at the right of a node IP following must be observed:

- The copy shall be at the right of IP, in this case ISIDE shall contain the character D in format A1(1HD).

- The copy shall be at the left of IP, in this case any character will be allowed except D or I.

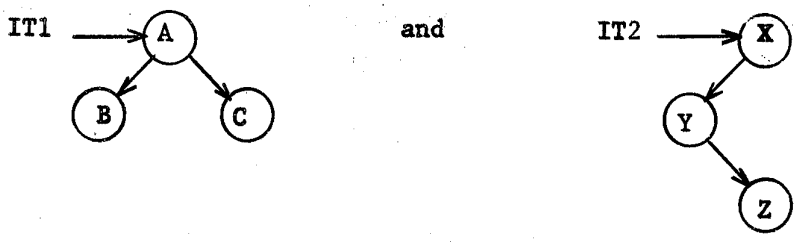
- IP and IQ may belong to the same tree, but both IP and IQ must belong to disjoint subtrees. If this rule is not followed the algorithm may fail,entering an infinite loop.

- IP may not be null.

The final tree is always normal (not threaded), even if IP and IQ belong to threaded binary trees the result will not be threaded.

IP may only be null if a completely new tree is to be obtained (ISIDE=1HI). If the copy is at the left or the right of a node IP, IP may not be null. Finally IP may point to any node. In the latter case the copied tree will be inserted between IP and the subtree of IP at ISIDE. IQ may not point to a head node.

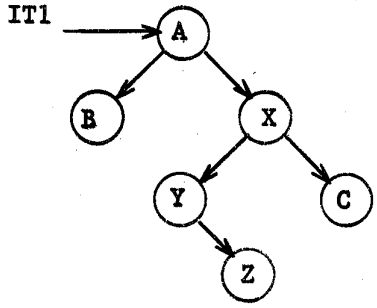
Given the trees:



The statement:

CALL COPIA (IT1, IT2, 1HD)

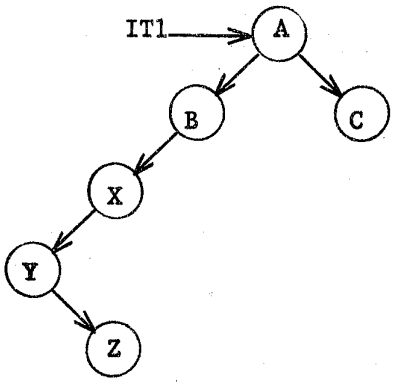
produces:



Using the same original trees IT1 and IT2 the statement

CALL CØPIA(IESQ(IT1),IT2,1H*)

Produces:



6 - DELETION OF TREES OR SUBTREES:

6.1 - Deletion of Complete Sections of a Binary Tree:

The subroutine DESLIG(IP) returns all the nodes to the available space (including IP), of the tree of subtree with root IP. IP may be any node, head, terminal or not terminal. If IP belongs to a threaded binary tree, the result is not a threaded binary tree. The user of this subroutine is responsible for the resetting of all pointers which point to the node IP.

6.2 - Deletion of Normal or Threaded binary Trees or Subtrees:

The subroutine DESATA(IP, SIDE) deletes the subtree at SIDE of

IP. IP may be any node, head, terminal or not terminal. IP may also belong to a threaded binary tree, and in this case the result will be a threaded binary tree. If SIDE contains the character D in format A1(1HD) the subtree on the right side of IP will be deleted. Any other character in SIDE will provoke a deletion of the subtree on the left side of IP. If IP points to a head node, the result will be a head node of an empty tree, regardless to the value of SIDE.

7 - CONVERSION OF NORMAL BINARY TREE TO THREADED BINARY TREE:

The function or subroutine IATAR(IP) changes a normal binary tree into a threaded binary tree. If IP is a null pointer, the subroutine will generate a head node. IP may be null(IP<1), a head node or a root. The tree with root IP, may be threaded or not. The value returned by the function, is the pointer to the head node of the transformed tree. The value of IP after execution will also point to the head of the transformed tree.

8 - TRAVERSING THREADED BINARY TREES:

The function ISSUS(IP, IQUE) returns a pointer to the sucessor node of IP in a traversing order IQUE. The node IP may be any, even head node, but IP must belong to a threaded binary tree.
IQUE is a code which determines which traversing order is to be taken. The value of IQUE should always belong to the range $1 \leq IQUE \leq 6$.

Codes of IQUE:

a - IQUE=1 traverse in right prefix order:

I - visit the root.
II - traverse the right subtree.
III - traverse the left subtree.

b - IQUE=2 traverse in left prefix order:

I - visit the root.
II - traverse the left subtree.
III - traverse the right subtree.

c - IQUE=3 traverse in right infix order:

I - traverse the right subtree.
II - visit the root.
III - traverse the left subtree.

d - IQUE=4 traverse in left infix order:

I - traverse the left subtree.
II - visit the root.
III - traverse the right subtree.

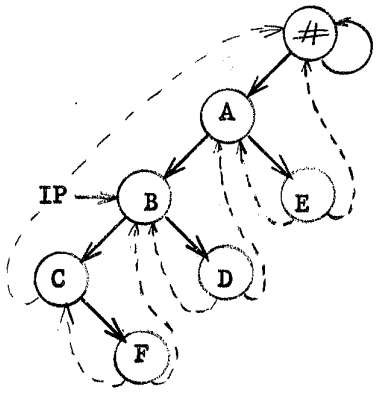
e - IQUE=5 traverse in right postfix order:

I - traverse the right subtree.
II - traverse the left subtree.
III- visit the root.

f - IQUE=6 traverse in left postfix order:

I - traverse the left subtree.
II - traverse the right subtree.
III- visit the root.

Given the tree



Statement	Result
IP1 = ISSUS(IP,1)	IP1→ D
IP1 = ISSUS(IP,2)	IP1→ C
IP1 = ISSUS(IP,3)	IP1→ F
IP1 = ISSUS(IP,4)	IP1→ D
IP1 = ISSUS(IP,5)	IP1→ A
IP1 = ISSUS(IP,6)	IP1→ E

9 - SCHEMATIC PRINTING OF A BINARY TREE:

The subroutine IMPARV(IP) prints a scheme of the tree with root IP. IP may be any node, e.g. head node, terminal or not terminal, and IP may also be a null pointer .

The node IP may belong to a normal or a threaded binary tree. If IP points to a head node, this node will be skipped and the printing will not contain the node IP, but will contain the tree of which IP is head. If IP is a null pointer, or IP points to a head of an empty tree, the message ARVØRE VAZIA will be printed.

The information field is printed in A6 format. It is the user's responsibility to mantain meaningful characters in the information field.

Each page of the printing will contain a maximum of 15 nodes. As many pages will be printed as are needed to list a complete tree. If a listing uses more then one page, conecting lines will be provided easing the concatenation of all pages. Any number of pages may be printed.

The level of the tree to be printed is limited to a height of 49. If this level is to small, it can be altered by defining a CØMMØN area in the main procedure. This CØMMØN area is:

```
CØMMØN /XPTO3/ ITAB(15,2),JMAX,ITBPT,INSAT(NN)
```

Where NN must be at least one unit greater then the highest level to be reached. The subprogram does not check the range of the subscripts. If the level reached is larger then NN malfunctioning of the system may result. The original version gives a dimension of 50 to NN.

10 - UTILITY FUNCTIONS:

Three functions can be used to collect pointers to the roots of all the trees which are in the working storage. A subroutine which prints the contents of the working storage is also available.

10.1 - Printing the Contents of the available Space:

The subroutine PRNPIL(CONVER) lists the contents of the working storage, including the nodes belonging to the free space stack. The number of nodes listed is variable, corresponding to the maximum number of nodes used at any time before the use of PRNPIL, this listing is terminated when an all null node is found, therefore the last node of the working storage must be null.

The information is printed in format A6. It is the user's responsibility to maintain meaningful characters in the information field. The conversion may be done before the calling of PRNPIL and in this case no argument should be given to PRNPIL. On the other hand the subroutine may convert the in - formation to format A6 during the run, in this case CONVER

should be the entry point of the conversion function. The argument given to the conversion function is the pointer to the node to be converted. The value returned by the conversion function must be in format A6. Nodes belonging to the available space stack do not provoke a call to the conversion function.

The list contains 55 nodes per page. Each line contains the node number (the pointer to the node), the format A6 value of the information field of the node, the contents of the unused 4 bits listed in binary, and also the left, the right link field and the available space tag.

10.2 - Collection of all Roots:

The subroutine or integer function, GARCØL collects the roots of all the trees still available in the working storage. This collection of roots occurs even when some of the trees, or subtrees, are not more refferenced by the user.

When GARCØL is used as a subroutine no argument is required when used as a function, a dummy argument is necessary. The argument is neither changed nor consulted.

For communication the subroutine GARCØL uses the area:

```
CØMMØN /LIXØ/ MAXNUM,IRØØTS(50)
```

Where:

MAXNUM - reveals the length of the vector IRØØTS. If MAXNUM is equal to zero when GARCØL is called, then the value of MAXNUM is set to 50.

IRØØTS - is a vector which contains a sequential list of the collected roots after returning from GARCØL. When a null pointer is found in IRØØTS there are no more roots, because the list is compacted before returning from GARCØL.

Due to the algorithm used, there may be more roots defined at times than there are trees actually available. This occurs because subtrees will be considered trees until a pointer to this subtree is found. Therefore the vector IRØØTS may run out of space even if only one tree is defined. A way to avoid this, is to try to restrict roots of trees to low order positions of the working storage.

If an amout other than 50 is to be allocated to IRØØTS, the alteration may be done by defining in the main procedure the necessary area:

```
CØMMØN /LIXØ/ MAXNUM,IRØØTS(NN)
```

Where NN is any integer constant. In this case MAXNUM must be set equal to this constant before invoking GARCØL.

If GARCØL is used as a function, the integer value retured is an error tag. If the returned value is zero no error has occured during the collection. If the value is odd, then IRØØTS run out of space.(Other values are described in 10.3) The error tags are accumulated and returned, indicating how many errors of a given type have occured.

The subroutine GARCØL searches for nodes in the working storage in ascending order. Whenever a node belonging to the available space stack or an actually collected node is found, the subroutine proceeds to the next node. When a node, which is neither available nor collected is found, it is assumed to be a root and all nodes beloging to this tree are collected by the subroutine TAG. When a head node is found by GARCØL only the tree pointed by this head is tagged.

10.3 - Tagging nodes Belonging to a Tree:

The subroutine or integer function TAG(IP) tags (collects) all the nodes belonging to the tree with root IP. TAG uses the area CØMMØN /LIXØ/ MAXNUM, IRØØTS(50). Each visited node is marked by TAG. If a marked node is found during the execution of TAG, the subroutine searches in the vector IRØØTS for a root equal to the pointer to the tagged node. If this pointer is found, then the tagged node is a root of a subtree of the tree being traversed. If no equal pointer is found, then there are

references from other trees to the same subtree; therefore a graph exists in the working storage. In the latter case an error message is printed and the value 1000 is added to the error tag. If a reference to a node belonging to the available space stack is found by TAG, an error message is printed and the value 1000000 is added to the error tag. The subroutine TAG may be used independently of GARCØL, but in so doing the user is responsible for maintaining meaningful pointers in the vector IRØØTS.

If TAG is used as a function, the value returned is the error tag. If the value is zero, then no error occured. If the value is not zero, then it represents the amount of errors which have occured. For each graph connection the value 1000 is added, and for each reference to the available space stack the value 1000000 is added.

10.4 - Printing the Collected Roots:

The subroutine or integer function PRIRAZ collects and prints all roots of trees still defined in the working storage. The collection of the roots in done by GARCØL. The contents of the area CØMMØN /LIXØ/ MAXNUM,IRØØTS(50) are available to the user after returning from PRIRAZ.

When using PRIRAZ as a function, the value returned is equivalent to that returned by GARCØL. If an error occured during the collection, then the value of the error flag will be printed by PRIRAZ.

11 - RESTRICTIONS:

a - The information of a node may be only a word long (36 bits).

b - The information field occupies the first line of the node, index equal to 1, IARV(1,XX).

c - If the information is to be printed by IMPARV, the information field should be in format A6 type.

d - The link fields together occupy one word. The processing of the link fields should be done exclusively by the pointer handling subroutines.

e - Any pointer with an algebraic value of less than 1 will be treated as a null pointer.

f - Thread lines are always null pointers. A thread line on the left link field points, taken in absolute value, to the predecessor node in left infix traversing order. A thread line on the right link field points, taken in absolute value, to the sucessor node in left infix traversing order.

g - Any node in which the information field contains a value greater than or equal to 34359738360_{10}, (377777777770_8) will be treated as a head node.

h - A head node always has a normal pointer at the right, and this pointer always points to the head node itself. The left pointer always points to the root of the binary tree which it, the head node, heads. If the tree is empty ,

the left pointer is a thread line pointing to the head itself.

i - All threaded binary trees must have one and only one head node.

12 - ERROR MESSAGES:

Error message identify the subroutine in which the error has occured. Some of the messages are terminal while others permit a resumption of processing.

***** DLIVRE - 1 ***** AREA 'BTREE' ESGOTADA.

This message is printed if more nodes are being required than there are available in the working storage CØMMØN /BTREE/. This message is terminal.

***** DLIVRE - 2 ***** NØDØ NNNNN NAØ E DE ESPACØ DISPØNIVEL.

This message is printed if a node NNNNN in the available space stack is not tagged available (4.1). This message is not terminal, an other node will be tried.

***** PLIVRE - 1 ***** NØDØ NNNNN JA PERTENCE AØ ESPACØ DISPØNIVEL

This message is printed if the node NNNNN is to be returned to the available space stack and it already belongs to the available space stack. This message is not terminal; it results in a 'no operation' of the subroutine.

***** DIRATA - 1 ***** NØDØ NNNNN E CABECA

This message is printed if the user tries to insert a node at the right side of the head NNNNN. This message is terminal.

***** ISSUS - 1 ***** CØDIGØ DE CAMINHAMENTØ ERRADØ

This message is printed if the traversing code IQUE given to ISSUS does not belong to the range 1<IQUE<6. This message is terminal.

***** GARCØL - 1 ***** LISTA DE RAIZES ESGØTADA.

This message is printed when more roots and/or subroots are being defined than there is space. The value 1 is added to the error tag and GARCØL returns to the calling point.

***** TAG - 1 ***** ENCØNTRADØ GRAFØ. NØDØ CØMMUM E NNNNN.

This message is printed whenever the subroutine TAG encounters a pointer to a common subtree. NNNNN is the root of the common subtree. This message is not terminal. The value 1000 is added to the error tag and processing continues.

***** TAG - 2 ***** LIGACAØ CØM PILHA LIVRE.

This message is printed whenever the subroutine TAG encounters a linkage to a node belonging to the available space stack. This message is not terminal. The value 1000000 is added to the error tag and processing continues.

ERRØ NA CØLECAØ - FLAG NNNNN

This message is printed by PRIRAZ if the value returned by GARCØL is not zero. NNNNN is the current value of the error flag.

13 - PROGRAMMING NOTES:

13.1 - CØMMØN Areas Used:

All the CØMMØN areas need not be declared by the user. Most of them are inter-subroutine communications areas. The CØMMØN areas are:

BTREE - the working area. It contains all the trees defined and the available space stack. May be enlarged by the user by a declaration CØMMØN /BTREE/ IARV(2,NN), where NN is the new size to be used (NN is normally equal to 100).

XPTO1 - is a communications area for the subroutine DLIVRE, PLIVRE and ESPACØ. It contains only one word and this word is a pointer to the top node of the free space stack. If this pointer is zero, the subroutine DLIVRE may look for more space increasing the upper bound of the used space. By the first calling do DLIVRE , the upper bound is set to 1 and the available space stack pointer is set to zero. If the available space stack

is initialized by a call to ESPACØ, then DLIVRE will not look for more if the available stack is empty.

XPTO2 - is a communications area for the subroutines IATAR and LIGAR. It contains only one word, the pointer to the last node visited by LIGAR.

XPTO3 - is a communications area of the subroutines for the schematic printing of a tree. It contains a descrip tive table of the page to be printed, during the printing of a tree.

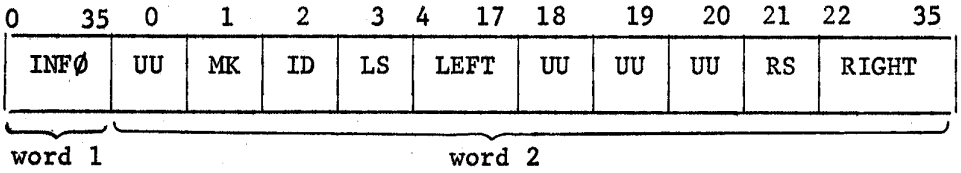
LIXØ - is a communication area for the subroutines GARCØL, TAG and PRIRAZ. It indicates the lenght of the root vector, and the root vector. After a root collection, the root vector contains in the low order positions all the roots found.

14 - ALGORITHMS:

14.1 - General Descriptions:

Since several references will be made to the fields of the node, by all the algorithms presented, and therefore I will describe the fields of a node only once. Throughout the text the names of the different fields will be refered to by the

names as follows.



where:

INFO - The information field (one word of 36 bits)

UU - Fields of one bit lenght each, which remain unused, and their value is not changed by any of the subroutines or functions.

LS and LEFT - The left link field. LS is the sign of the link, and LEFT is the numerical value of the link (15 bits together).

RS and RIGHT - The right link field. RS is the sign and RIGHT is the numerical value of the link (15 bits together).

ID - Is a tag which indicates if the nodes belongs to the available space stack (ID=1) or not (ID=0). (One bit).

MK - Is a tag which indicates whether a node was marked (TG=1) or not (TG=0)during the root collection (One bit).

14.2 - Obtaining of a Node From the Available Free Space DLIVRE

DL - This subroutine returns a pointer ID to an available node. The subroutine uses UPB, the greatest value reached by IP. A free space stack is used, the top of this stack is pointed by FPT. The values of UPB and FPT are set to zero by the first call to DLIVRE.

DL1 - (Stack empty) If FPT>0 go to step DL3. Otherwise if FPT<0 go to step DL4.

DL2 - (Generate more space) Set UPB ← UPB+1. If now any of the fields of NODE(UPB) are not zero go to step DL4.* Otherwise set IP←UPB and return to the calling point.

DL3 - (Pop up the stack) Set IP←FPT and FPT←RIGHT(FPT). If now ID(IP)=0 print an error message and go back to step DL1. Otherwise clear all fiedls of NODE(IP) and return to the calling point.

DL4 - (Memory overflow) Print an error message and stop.

14.3 - Return a Node to the Available Space Stack, PLIVRE:

PL - This subroutine receives a pointer IP to a node which should be returned to the free space stack. It uses the free space stack and the pointer to its top FPT.

---

*Under IBSYS (IBM-7044) all variable areas, CØMMØN, DIMENSION or others, are initialized zero when loaded by IBLDR.

PL1 - (Test null pointer) If IP<1 return to the calling point.

PL2 - (Test if the node is already free) If ID(IP)=1 print an error message and return to the calling point.

PL3 - (Turn node free) Set RIGHT(IP)←FPT, ID(IP)←1, FPT←IP and return to the calling point.

14.4 - Initialization of the Free Space, ESPACØ:

ES - This subroutine receives as argument a number N which indicates the number of nodes in the working storage. The free space stack is generated without considering any information in the working storage.

ES1 - (Inhibit reset of DLIVRE) Execute DLIVRE. (This step must be done to avoid an undesired reset made by the first call to DLIVRE).

ES2 - (Create stack) Set RIGHT(I)←I+1 and ID(I)←1 for all $1 \leq I \leq N-1$. Set RIGHT(N)← -N and ID(N)←1, afterwards return to the invoking point.

14.5 - Insertion of a Node into a Threaded Binary Tree:

a - ESQATA

EA - This subroutine receives the arguments IP and IQ. It inserts the node pointed by IQ as a new root of the subtree of the node pointed by IP. It uses the function ISSUS to place the sucessor in right infix traversal ($I).

EA1 - (Adjust links) Set LEFT(IQ)←LEFT(IP), RIGHT(IQ)← -IP and LEFT(IP)←IQ.

EA2 - (Correct thread line if the left subtree is not empty) If LEFT(IQ)<1 return to the calling point. Otherwise obtain $IQ ($IQ - sucessor in right infix traversal) and set RIGHT($IQ)← -IQ, then return to the calling point.

b - DIRATA

DA - The subroutine DIRATA inserts a node pointed by IQ as a new root of the right subtree of the node pointed by IP. Both IP and IQ are given as arguments. It uses the function ISSUS to get the sucessor node in left infix traversal. (I$).

DA1 - (Test if IP is head) If INFØ(IP)≥ 34 359 738 360 print an error message and stop.

DA2 - (Adjust links) Set RIGHT(IQ)← RIGHT(IP), LEFT(IQ)← -IP and RIGHT(IP)←IQ.

DA3 - (Correct thread line if right subtree is empty) If RIGHT(IQ)<1 return to the calling point. Otherwise obtain IQ$, set LEFT(IQ$)← - IQ and return to the calling point.

14.6 - Copy of a Tree, CØPIA

CY - This subroutine receives the arguments IP, IQ and SIDE. IQ is the pointer to the root of the tree to be copied. SIDE indicates whether the copy should be a completely new tree (SIDE=1HI), or if the copy should be made preceeding the root of the right sub tree (SIDE=1HD), or the left subtree (SIDE≠1HI and ≠1HD) of the node pointed by IP.

This subroutine uses the temporaries IR, IP1 and IP2. It also uses a stack, pointed by the node TP. This stack is initially empty (TP = 0). Each node of the stack contains 2 fields. IP1S and IQS. The field IQS contains a pointer to a node of which the right subtree was not completely copied. IP1S points to the copy of the root of the right subtree, corres - ponding to the node pointed by IQS.

CY1 - (Test action) If SIDE contains the character I go to step CY10. Other wise set IR←IP.

CY2 - (Test if the tree to be copied is empty) If IQ<1 return to the calling point.

CY3 - (Test if the copy is at the right of IP) If SIDE contains the character D go to step CY10.

CY4 - (Copy a node to the left) Obtain a node IP2 from the free space. Set INFØ(IP2)←INFØ(IQ), LEFT(IP2)←LEFT (IP), RIGHT(IP2)←0 and LEFT(IP)←IP2.

CY5 - (Copy a node to the right) If RIGHT(IQ)<1 go to step CY6. Otherwise obtain a node IP1 from the free space Set INFØ(IP1)←INFØ(RIGHT(IQ)), RIGHT(IP1)←RIGHT(IP), LEFT(IP1)←0 and RIGHT(IP)←IP1.

CY6 - (Push down) If LEFT(IQ)<1 go to step CY7. Otherwise set TP←TP+1, IQS(TP)← IQ and IP1S(TP)← IP1. Then set IP← IP2 and IQ← LEFT(IQ) and go back to step CY4.

CY7 - (Copy the right subtree of IQ) If RIGHT(IQ)< 1 go to step CY8. Otherwise set IP2← IP1, IP← IP1 and IQ← RIGHT (IQ) and go back to step CY5.

CY8 - (Pop up) If TP=0 set IP← IR and return to the calling point. Otherwise set IQ← IQS(TP), IP1← IP1S(TP) and TP← TP-1. Then go back to step CY7.

CY9 - (Start copy to the right) Obtain a node IP2 from the free space. Set INFØ(IP2) ←INFØ(IQ), RIGHT (IP2)← RIGHT(IP), LEFT(IP2)← 0 and RIGHT(IP)← IP2. Then start normal copying at step CY5.

CY10 - (Prepare a root node for the copy) Obtain a node IP2 from the free space. Set IR← IP2. (This guarantees that the value returned by IP points to the root of the copy tree).Set INFØ(IP)← INFØ(IQ), RIGHT (IP)← 0, LEFT(IP) ←0 and then start normal copying at step CY5.

14.7 - Deletion of a Tree, DESLIG:

DS - This subroutine receives an argument IP, which points to the root of a tree to be completely deleted. A stack is used to permit the posfix traversal of the tree. Each node of the stack contains 2 fields PT and TAG. PT points to a visited node, TAG indicates whether the right or the left subtree is beeing traversed. The pointer to the top of this stack is TP, and the stack is initially empty. Finally it uses the subroutine PLIVRE. (A node is returned to the available space stack only after the left and the right subtree of this node have been completely deleted.)

DS1 - (Test end.) If IP< 1 go to step DS3.

DS2 - (Step left) Set TP← TP+1,PT(TP)← IP, TAG(TP)← 0, IP← LEFT(IP) and go back to step DS1.

DS3 - (Step right) If TP=0 return to the calling point. Otherwise if TAG(TP)=0 set TAG(TP)← 1, IP← RIGHT (PT(TP)) and go back to step DS1.

DS4 - (Delete node and pop-up) Set IP← PT(TP) and TP← TP-1 Execute PLIVRE(IP), and go back to step DS3.

14.8 - Deletion of a Subtree, DESATA:

DA - This subroutine receives a pointer IP and an action driver SIDE. It deletes a subtree of the node pointed by IP. The subtree is placed at SIDE of the node IP. The subroutine uses the temporary IQ and the subroutine DESLIG for deletion of the subtree.

DA1 - (Test if head) If INFØ(IP)≥ 34359738360 go to step DA4. Otherwise set IQ← IP. Afterwards if SIDE contains the character D go to step DA3.

DA2 - (Delete left subtree) If IQ≥ 1 set IQ← LEFT(IQ) and repeat this step. Otherwise execute DESLIG(LEFT(IP)) Afterwards set LEFT(IP)← IQ and return to the calling point.

DA3 - (Delete right subtree) If IQ≥ 1 set IQ← RIGHT(IQ) and repeat this step. Otherwise execute DESLIG(RIGHT(IP)).

Afterwards set RIGHT(IP)← IQ and return to the calling point.

DA4 - (Delete the complete tree) Execute DESLIG(LEFT(IP)). Afterwards set LEFT(IP)← -IP and return to the calling point.

14.9 - Conversion of a normal binary tree into a threaded binary tree:

This set contains 2 subroutines. They use a communication area where a pointer IL, to the last altered node, is stored.

a - IATAR

IA - This subroutine or function is the leading subprogram of the set. It initializes and ends the conversion and if a head node is needed it creates one. As an input argument this subroutine receives a pointer IP to the root of the tree to be converted. As an output argument the same pointer (IP) is altered to point to the head of the transformed tree. The subroutine IATAR uses the subroutine LIGAR for the conversion. Finally it uses a temporary IP1.

IA1 - (Test null pointer) If IP< 1 go to step IA2. Otherwise set IP1←IP. If INFØ(IP)≥ 34359738360 go to step IA3.

IA2 - (Create head) Obtain a node IP1 from the available space. Set INFØ(IP1)← 34359738360. If IP<1 set IP← -IP1. Set LEFT(IP1)←IP.

IA3 - (Adjust links) Set RIGHT(IP1) and IP← IP1. If now LEFT(IP)<1 return to the calling point. Otherwise set IL←IP1, execute LIGAR(LEFT(IP1)). Afterwards set RIGHT(IL)← -IP1 and return to the calling point.

b - LIGAR

LG - This subroutine converts a normal binary tree into a threaded binary tree. The last right thread line (ponting to the head) is made by IATAR. LIGAR uses an auxiliary stack pointed by TP. The stack is initially empty (TP=0). Each node of the stack contains 2 fields One, IPS, points to a visited node. The other one, TAG, indicates wether the left (TAG=0) or the right (TAG=1) subtree was entered. Further a temporary ITAG is used.

LG1 - (Push down) If LEFT(IP)<1 go to step LG2. Otherwise set TP←TP+1, IPS(TP)←IP, TAG(TP)←0, IP←LEFT(IP) and repeat this step.

LG2 - (Create a left thread line) Set LEFT(IP)← -IL.

LG3 - (Mark threaded node) Set IL←IP. If RIGHT(IP)<1 go to step LG4. Otherwise set TP←TP+1, IPS(TP)←IP, TAG(TP)←1, IP←RIGHT(IP) and go back to step LG1.

LG4 - (Pop up) If TP=0 return to the calling point. Otherwise set IP←IPS(TP). ITAG←TAG(TP), TP←TP=1. If now ITAG=1 repeat this step.

LG5 - (Create a right thread line) Set RIGHT(IL)← -IP and go back to step LG3.

14.10 - Traversing threaded binary trees, ISSUS

The function ISSUS contains 3 distinc functions. The function receives the arguments IP and IQUE. IP points to a node. IQUE indicates which traversing form should be used to obtain the sucessor node of IP. ISSUS (the value of the function) is the pointer to the sucessor node of IP, after complete execution. The value of IP and IQUE are not destroyed by this function . Finally a temporary IP1 is used.

a - Right (IQUE=1) and left (IQUE=2) prefix traversal.

PR1 - (Test head) Set IP1←IP. If INFØ(IP)≥34359738360 set ISSUS←|LEFT(IP)| and return to the calling point.

PR2 - (Search subtree) If IQUE=1 set ISSUS←RIGHT(IP). If IQUE=2 set ISSUS←LEFT(IP). If now ISSUS≥1 return to the calling point. Otherwise set ISSUS←IP.

PR3 - (Move to a lower level) If IQUE=1 set IP1←LEFT(ISSUS) If IQUE=2 set IP1←RIGHT(ISSUS). Now set ISSUS←|IP1|.

PR4 - (Test if found) If now INFØ(ISSUS)<34359738360 and IQUE=1 return to the calling point. Otherwise if IP1<1 go back to step PR3. Otherwise return to the calling point.

b - Right (IQUE=3) or left (IQUE=4) infix traversal.

IN1 - (Test thread line) If IQUE=3 set IP1← LEFT(IP). If IQUE=4 set IP1←RIGHT(IP). Now set ISSUS←|IP1|. If IP1<1 return to the calling point.

IN2 - (Search thread line) If IQUE=3 set IP1←RIGHT(ISSUS). If IQUE=4 set IP1←LEFT(ISSUS). If now IP1<1 return to the calling point. Otherwise set ISSUS←IP1 and repeat this step.

c - Right (IQUE=5) or left (IQUE=6) posfix traversal.

PØ1 - (Test head) Set ISSUS←IP. If INFØ(IP)≥34359738360 go to step PØ6.

PØ2 - (Search lower level) If IQUE=5 set ISSUS←LEFT(ISSUS). If IQUE=6 set ISSUS←RIGHT(ISSUS). If now ISSUS<1 go to step. Otherwise repeat this step.

P∅3 - (Test if father found) Set ISSUS←|ISSUS|. If IQUE=5 set IP1←RIGHT(ISSUS). If IQUE=6 set IP1←LEFT(ISSUS). If now IP1=IP go to step P∅5.

P∅4 - (Search father) Set ISSUS←IP1. If IQUE=5 set IP1←LEFT(ISSUS). If IQUE=6 set IP1←RIGHT(ISSUS). If now IP1=IP return to the calling point. Otherwise repeat this step.

P∅5 - (Search brother) If IQUE=6 and INF∅(ISSUS)≥34359738360 return to the calling point. Otherwise if IQUE=5 set IP1←LEFT(ISSUS). If IQUE=6 set IP1←RIGHT(ISSUS) If now IP1<1 return to the calling point.

P∅6 - (Search general terminal or head) If IQUE=5 set IP1←ISSUS(ISSUS,3). If IQUE=6 set IP1←(ISSUS,4). Now set ISSUS←IP1. If now LEFT(ISSUS)<1 and RIGHT(ISSUS)<<1 or if INF∅(ISSUS)≥34359738360 return to the calling point. Otherwise repeat this step.

14.11 - Schematic printing of a binary tree

This set of subroutines is formed by 4 subroutines IMPARV, INIPRN, PRTR and PRINAT. The subroutines use a communication area containing LMAX, TBP, the vectors ND, NL, INSAT. The vector ND contains pointers to nodes which are to be printed on the actual page. The vector NL indicates the level of the node. ND and NL have an one to one correspondence, both contain 15 elements at all. Further LMAX indicates the maximum level

any node had until now, and, finally TBP contains the index of the last node inserted into ND and LV. INSAT is used solely by PRINAT. It indicates wether all conectors on the preceeding page where completed or not.

a - IMPARV.

IV - This subroutine is the header routine of this set. It initializes the vectors ND and LN setting them to zero, also LMAX and TBP are set to zero. Finally the vector INSAT is set to zero. The subroutine receives a pointer IP to the root of the tree to be printed. It uses the subroutines PRTR, INIPRN and PRINAT, and the temporary I.

IV1 - (Test empty tree) If IP<1 go to step IV4. Otherwise if INFØ(IP)≥ 34 359738360 set I←LEFT(IP). Now again, If I<1 go to step IV4.

IV2 - (Reset) Call INIPRN to set the contents of ND, LN, INSAT, LMAX and TBP to zero.

IV3 - (Print) Execute PRTR(I,1) Afterwards if TBP≠0 print the last page calling PRINAT, and then return to the calling point.

IV4 - (Empty tree) Print the message "ARVORE VAZIA" signalizing that the tree pointed by IP is empty and return to the calling point.

b - PRTR

PR - This subroutine receives 2 arguments IP and LV . The first points to the root of the tree to be printed. LV indicates the level of the node. The subroutine uses a stack pointed by TP. This stack is initially empty (TP=0). Each node contains 3 fields IPS, LVS and TAG. IPS contains a pointer to a visited node. TAG indicates wether the left or the right subtree of IPS is beeing traversed. Finally LV indicates the level of the visited node IPS. Further the subroutine uses a temporary ITAG.

PR1 - (Push down) If LEFT(IP)<1 go to step PR2. Otherwise set TP←TP+1, IPS(TP)←IP, LVS(TP)←LV, TAG(TP)←0, IP←LEFT(IP) ,LV←LV+1 and repeat this step.

PR2 - (Insert a new node into the vectors) Set TBP←TBP+1, ND(TBP)←IP, LN(TBP)←LV. If LV>LMAX set LMAX←LV+1.

PR3 - (Print a page) If TBP<15 go to step PR4. Otherwise execute PRINAT printing a page described by ND and LN. Afterwards set the contents of the vectors ND and LN to zero and finally set TBP←0.

PR4 - (Push down) If RIGHT(IP)<1 go to step PR5. Otherwise set TP←TP+1, IPS(TP)←IP, LVS(TP)←LV, TAG(TP)←1, IP←RIGHT(IP), LV←LV+1 and go back to step PR1.

PR5 - (Pop up) If TP=0 return to the calling point. Otherwise set IP←IPS(TP), LV←LVS(TP), ITAG←TAG(TP)and TP←TP-1. If now ITAG=0 go back to step PR2. Otherwise repeat this step

c - PRINAT

PT - This subroutine prints a page of the tree scheme. It uses the vectors LINE1 and LINE2, which will contain the conective lines and the information to be printed. Further LEFTMARGIN and RIGHTMARGIN are used to contain conective lines if a conection is made to an other page. The variable I counts the number of nodes inserted into one line. The variable N counts the level of the line to be printed. (Nodes which have the same level will be printed on the same line or on a corresponding line on an other page). I1 is a pointer to the leftmost position a node may occupy. Finally the variables J and I2 are temporaries.

PT1 - (Global reset) Skip to a new page and set N←1.

PT2 - (Line reset) Clean LEFTMARGIN, LINE1, RIGHTMARGIN and LINE2. Set I←1, If INSAT(N)=0 go to step PT8.

PT3 - (Search node) If LN(I)=N go to step PT6. LN(I)=N+1 go to step PT7. Otherwise set I←I+1. If now I≤15 repeat this step.

PT4 - (No node was found) Insert '------' into both LEFTMARGIN and RIGHTMARGIN and fill the whole LINE1 with '------'.

PT5 - (Print and count) Printout: LEFTMARGIN, LINE1, RIGHTMARGIN and LINE2. Set N←N+1. If now N>LMAX return to the calling point. Otherwise go back to step PT2.

PT6 - (Son was printed on a preceeding page) Insert '------' into LEFTMARGIN. Insert '--X---' into LINE1(I). If I>2 fill LINE1(J) with '------' for all J, 1≤J≤I-1. Set INSAT(N)←0. Insert INFØ(ND(I)) into LINE2(I). Set I1←I and go to step PT15.

PT7 - (Father was printed on a preceeding page) Insert '------' into LEFTMARGIN. Insert '--V ' into LINE1(I) If I>2 fill LINE1(J) with '------' for all J, 1≤J≤I - 1. INSAT(N)←0 and go to step PT9.

PT8 - (Search father) If LV(I)=N go to step PT12.

PT9 - (Count colunm) Set I←I+1. If now I≤15 go back to step PT8. Otherwise set I←I1.

PT10 - (Search son referenced from the nex page) If I>15 go back to step PT5. Otherwise if LN(I)=N+1 go to step PT11. Otherwise set I←I+1 and repeat this step.

PT11 - (Insert refference from nextpage) Insert '------' into RIGHTMARGIN. Set INSAT(N)←1. Insert ' V---' into LINE1(I). If I≤15, fill LINE1(J) with '------' for all J, I+1≤J≤15. Afterwards go back to step PT5.

PT12 - (Test if left subtree is empty) Insert INFØ(ND(I)) into LINE2(I). If LEFT(ND(I))<1 set I1←I+1 and go to step PT15. Otherwise set I2←I1

PT13 - (Search left son) If LN(I2)=N+1 go to step PT14. Otherwise set I2←I2+1 and repeat this step.

PT14 - (Set left conection lines) Insert '--X---' into LINE1(I). Insert ' V---' into LINE1(I1). If I1≤I-1, fill LINE1(J) with '------' for all J, I1+1≤J≤I-1. Afterwards set I1←I+1.

PT15 - (Test if right subtree is empty) If RIGHT(ND(I))<1 go back to step PT9. Otherwise insert '--X---' into LINE1(I).

PT16 - (Search right son) If I1>15 go to step PT18. If LN(I)=N+1 go to step PT17. Otherwise set I1←I1+1 and repeat this step.

PT17 - (Set right conection lines) Insert '--V ' into LINE1 (I1). If I≤I1-1 fill LINE1(J) with '------' for all J, I+1≤J≤I1-1 and go back to step PT9.

PT18 - (Connect father with right margin) Insert '------' into Rightmargin. Set INSAT(N)←1. If I+1≤15 fill LINE1(J) with '------' for all J, I+1≤J≤15. Afterwards go back to step PT5.

14.12 - Printing the contents of the working storage, PRNPIL.

PP - This subroutine prints the contents of the working storage. The nodes are printed in ascending order. The algorithm terminates when a node with all its fields equal to zero is found. The subroutine uses an external function EXFUN for converting the information field into format A6. This function may or not be transmitted as an argument. In the case no argument is transmitted, the subroutine PRNPIL assumes that the contents of the information field is already in format A6. In the case 1 argument is transmitted, PRNPIL calls EXFUN (transmitted argument) for the conversion. The subroutine uses the node counter I, the subroutine NARG to obtain the number N of arguments and the pointer FPT which points to the top of the available space stack.

PP1 - (Initialize) Set I←0, execute NARG obtaining the number N of arguments of the call to PRNPIL.

PP2 - (Conversion of the information field) If ID(I)=1 or if N=0 go to step PP3. Otherwise execute the transmitted function EXFUN obtaining the conversion

of the information field of the node I.

PP3 - (Printing the contents of a node) If MØD(I,55)=1 print the title of the page. Now print the value of I, the converted information field, the contents of the unused bits, the left and the right link fields and, finally, the available stack tag ID of the node I.

PP4 - (Count nodes and finalize) Set I←I+1. If the contents of all fields of the node I, are equal to zero, go to step PP5. Otherwise return to step PP2.

PP5 - (Print the top pointer of the available space stack) If FPT 0 print 'PILHA LIVRE VAZIA', returning afterwards to the calling point. Otherwise print 'PILHA LIVRE' and the current value of FPT, returning afterwards to the calling point.

14-13 - Collection of all roots

a - Collection subroutine GARCØL

GC - This subroutine visits all nodes in the working storage in ascending order. The procedure stops when an all zero node is found in the working storage. When a node is found, which does neither belong available space stack, nor is marked as visited before, it is assumed to be a root of a tree. The subroutine

TAG is invoked to mark all nodes belonging to this particular tree. After the marking the pointer to the root is introduced into the IRØØTS vector. The subroutine GARCØL uses a counter I pointing to nodes in the working storage, a counter J pointing to nodes in the IRØØTS vector, an error flag EF and the number MAXNUM which indicates the number of nodes in IRØØTS.

GC1 - (Initialize) Set EF←0. If MAXNUM=0 set MAXNUM← 50. Now set IRØØTS(J)←0 for all 1≤J≤MAXNUM. Finally set I←0.

GC2 - (Count nodes and finalize) Set I←I+1. If now all fields of node I are equal to zero go to step GC6.

GC3 - (Test if root found) If MK(I)≠0 or ID(I)≠0 go back to step GC2.

GC4 - (Mark the tree and insert the pointer into IRØØTS) If INFØ(I)≥34 359 738 360 then execute TAG only for the left subtree of node I. Otherwise execute TAG for the tree with root I. Add the error flag returned by TAG to EF. Now find a free slot J in the IRØØTS vector (This means: find a J such that IRØØTS(J)=0) If none such J exist then go to step GC5. Otherwise set IRØØTS(J)←I and return to step GC2. (Head nodes are not marked. Therefore any head node used erronously in a tree will provoke a TAG-1 error message).

GC5 - (Error, too many roots are defined) Print the error message GARCØL - 1 and set EF←EF+1.

GC6 - (Compact nodes of IRØØTS) Rearange the contents of IRØØTS so that no null pointer is mixed with root pointers. (This will compact all root pointers to the lower adresses of IRØØTS. All the remainding nodes of IRØØTS will be zero).

GC7 - (Remove markings) Set NK(J)←0 for all $1 \leq J \leq I$ and then return to the calling point. (This step resets the nodes, enabling therefore, any later call to GARCØL to collect all trees again).

b - Tagging subroutine TAG.

TG - This subroutine traverses a tree with root IP. The pointer IP is transmitted as an argument to TAG. Each visited node is marked. If an already marked node is found, the subroutine searches for the root in the IRØØTS vector. If no equal pointer is found, an error message is printed since a common subtree (graph) was found. The subroutine uses a stack TØP containing pointers PST to nodes which right subtrees still must be traversed, The counter J is used to point to a node in the IRØØTS vector. EF is an error flag and MAXNUM is the amount of nodes available in IRØØTS.

TG1 - (Initialize) Set EF←0 and TØP←0.

TG2 - (Test action) If IP<1 go to step TG4. If MK(IP)≠0 go to step TG5. If ID(IP)≠0 go to step TG7.

TG3 - (Mark and push down) Set MK(IP)←1, TØP+1, PST(TØP)←IP, and IP←LEFT(IP). Afterwards go back to step TG2.

TG4 - (Pop up) If TØP=0 return to the calling point. Set IP←RIGHT(PST(TØP)), TØP←TØP-1 and go back to step TG2.

TG5 - (Search marked subtree) Find a J such that IRØØTS(J) = IP for any 1≤J≤MAXNUM. If none such J exist go to step TG6. Otherwise set IRØØTS(J)←0 and go back to step TG4.

TG6 - (Graph found) Print IP and the error message TAG-1 Set EF←EF+1000. And go back to step TG4.

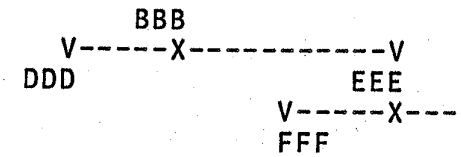
TG7 - (Linkage to the available space stack found) Print the error message TAG - 2. Set EF←EF+1000000 and go back to step TG4.

15 - BIBLIOGRAPHY:

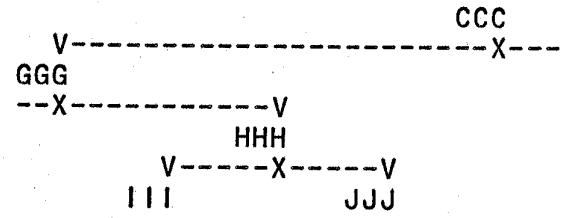
1 - Knuth, D. E. ; "The Art of Computer Programming"; Chapter 2. Addison Wesley; 1968.

16. **APPENDIX:**

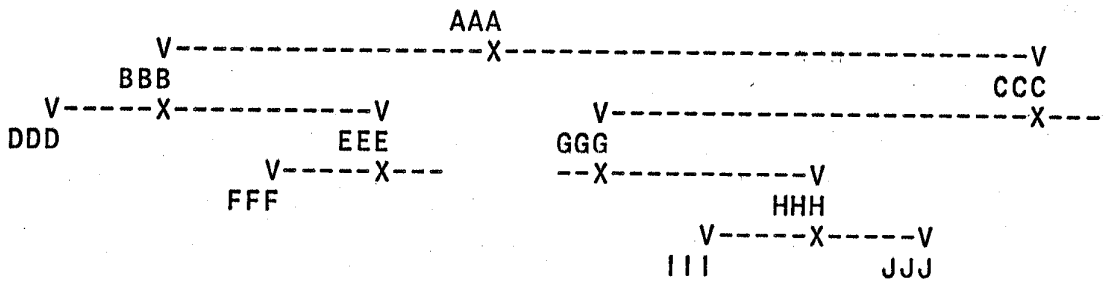
```
 DATA A,B,C,D,E,F,G,H,I,J /3HAAA,3HBBB,3HCCC,3HDDD,3HEEE,3HFFF,
* 3HGGG,3HHHH,3HIII,3HJJJ /

 I1=ICNS(B,ICNS(D,0,0),ICNS(E,ICNS(F,0,0),0))
 CALL IMPARV(I1)
```




```
I2=ICNS(C,ICNS(G,0,ICNS(H,ICNS(I,0,0),ICNS(J,0,0))),0)
CALL IMPARV(I2)
```




```
I3=ICNS(A,I1,I2)
 CALL IMPARV(I3)
```

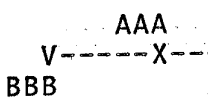


```
CALL PRNPIL

NODO    INFO.    BITS    IESQ    IDIR    TAG

   1    DDD      0000       0       0      0
   2    FFF      0000       0       0      0
   3    EEE      0000       2       0      0
   4    BBB      0000       1       3      0
   5    III      0000       0       0      0
   6    JJJ      0000       0       0      0
   7    HHH      0000       5       6      0
   8    GGG      0000       0       7      0
   9    CCC      0000       8       0      0
  10    AAA      0000       4       9      0


FUNCTION MODIF(I)
COMMON /BTREE/ IARV(2,100)
DATA ICAB,ICA /0377777777770,6HCABECA/
MODIF=IARV(1,I)
 IF(MODIF.GE.ICAB) MODIF=ICA
RETURN
END

I4=IATAR(I3)
CALL PRNPIL(MODIF)

NODO    INFO.    BITS    IESQ    IDIR    TAG

   1    DDD      0000     -11      -4      0
   2    FFF      0000      -4      -3      0
   3    EEE      0000       2     -10      0
   4    BBB      0000       1       3      0
   5    III      0000      -8      -7      0
   6    JJJ      0000      -7      -9      0
   7    HHH      0000       5       6      0
   8    GGG      0000     -10       7      0
   9    CCC      0000       8     -11      0
  10    AAA      0000       4       9      0
  11    CABECA   0000      10      11      0
```

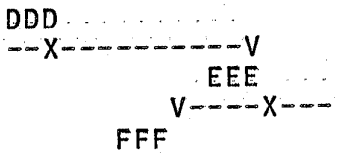
```
    PRINT 101
101 FORMAT(47H1 NRO    PRFD    PRFE    INFD    INFE    POSD    POSE,//)
    DO 1 J=1,6
  1 NODO(J)=14
    DO 2 I=1,15
    DO 3 J=1,6
    NODO(J)=ISSUS(NODO(J),J)
  3 LINHA(J)=MODIF(NODO(J))
  2 PRINT 100,I,LINHA
100 FORMAT(I5,6(3X,A4))
```

NRO	PRFD	PRFE	INFD	INFE	POSD	POSE
1	AAA	AAA	CCC	DDD	JJJ	DDD
2	CCC	BBB	JJJ	BBB	III	FFF
3	GGG	DDD	HHH	FFF	HHH	EEE
4	HHH	EEE	III	EEE	GGG	BBB
5	JJJ	FFF	GGG	AAA	CCC	III
6	III	CCC	AAA	GGG	FFF	JJJ
7	BBB	GGG	EEE	III	EEE	HHH
8	EEE	HHH	FFF	HHH	DDD	GGG
9	FFF	III	BBB	JJJ	BBB	CCC
10	DDD	JJJ	DDD	CCC	AAA	AAA
11	CABE	CABE	CABE	CABE	CABE	CABE
12	AAA	AAA	CCC	DDD	JJJ	DDD
12	CCC	BBB	JJJ	BBB	III	FFF

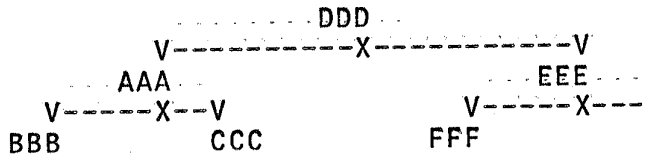
```
I1=ICNS(A,ICNS(B,0,0),ICNS(C,0,0))
I2=ICNS(D,0,ICNS(E,ICNS(F,0,0),0))
CALL IMPARV(I1)
```



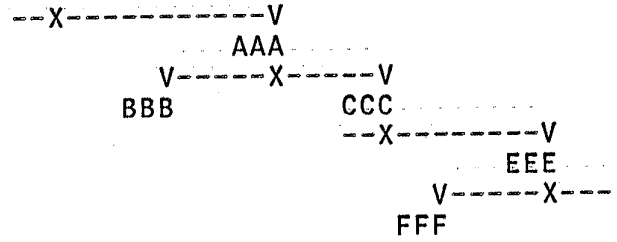

```
CALL IMPARV(I2)
```




```
CALL COPIA(I2,I1,1H*)
CALL IMPARV(I2)
```




```
CALL DESATA(I2,1HE)
CALL COPIA(I2,I1,1HD)
CALL IMPARV(I2)
```



```
CALL ESPACO(7)
 I5=0
I5=IATAR(I5)
 I1=ICNS(A,0,0)
I2=ICNS(B,0,0)
 I3=ICNS(C,0,0)
 CALL ESQATA(I5,I1)
CALL DIRATA(IESQ(I5),I2)
CALL ESQATA(I1,I3)
CALL IMPARV(I5)

      AAA
   V----X-----V
 CCC         BBB


CALL PRNPIL(MODIF)


NODO    INFO.    BITS    IESQ    IDIR    TAG

   1    CABECA   0000       2       1      0
   2    AAA      0000       4       3      0
   3    BBB      0000      -2      -1      0
   4    CCC      0000      -1      -2      0
   5    000000   0000       0       6      1
   6    000000   0000       0       7      1
   7    000000   1000       0       7      1

PILHA LIVRE   5


CALL DESATA(IESQ(I5),1HD)
CALL IMPARV(I5)

     AAA
   V----X---
CCC
```