BALANCED BINARY TREE CODE FOR SCIENTIFIC APPLICATIONS

S.C. PARK and J.P. DRAAYER

Department of Computer Science and Department of Physics and Astronomy, Louisiana State University,
Baton Rouge, LA 70803-4601, USA

Received 30 March 1969

A set of easy-to-use FORTRAN routines for building and accessing data structures of the tree type commonly encountered in scientific applications is presented. Push and insert times are $O(\log n)$, where $n$ is the number of elements in the list. The routines implement AVL or height-balanced binary tree logic. Each tree is a linear integer array. The first ten elements of a tree array specify its structure and the remaining elements are dedicated to node information. Each node includes key and integer data elements in addition to the necessary linkage information. The keys and data can each be multiple word atoms. The routines are generic so even though the structure of any one specific tree is fixed, an application can include several trees that have different structures with no additional code requirements. An example that illustrates how the application can be integrated into an existing code is included.

PROGRAM SUMMARY

Type of program: BITREE

Catalog number: ABIR

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Computer: IBM 3081/6000, DEC VAX-11/780, IBM PC/XT

Operating system: MS/DOS 4.5, DOS 3.2, respectively

Programming language used: FORTRAN

High speed storage required: 670 bytes, 1860 bytes, 465 bytes, respectively, for the BITREE routines. Main program and tree storage requirements are over and above this amount

Preliminary: none

No. of lines in program: 386 for the BITREE routines. The sample program, which includes the BITREE routines, is 644 lines of code

Keywords: binary tree, dynamic storage, AVL tree

* Supported in part by a grant from the US National Science Foundation.

Nature of the physical problem:

Typical scientific programming applications require numerous calls to one or more subroutines. The intermediate results generated by these calls are usually not saved; if the same information is required at a later stage it is simply recomputed. While wasteful of cpu power, this mode of operation is attractive because it spares the user the time and effort associated with the development of complex data storage and retrieval algorithms. However, if the number of redundant calls to a particular subroutine is large and the time spent in that subroutine amounts to a sizable fraction of the total cpu time, this simplistic programming style must yield to more sophisticated practices that allow for better utilization of the available machine resources.

This routine provided motivation for the development of BITREE which is a binary tree adjunct for scientific applications. Specifically, the objective was to develop a set of low-overhead and easy-to-use routines that allow one to build data structures with entities that can be re-entered and/or changed as necessary to optimize program execution times against machine memory capacity. The following were considered essential features:

1. Simple – the routine should be integrable into existing code with a minimum amount of effort on the part of the user.

2. Efficient – the routine should require little storage and apply minimal change to both simple and complex data structures.

0010-4655/89/$02.50 © Elsevier Science Publishers B.V. (North-Holland Physics Publishing Division)
A highly balanced binary tree solves the problem. It gives \( F(n) \) time for insert and delete operations on dynamically generated sets. A full description of the theory behind the splay tree concept, common referred to as AVL tree logic after Adelson-Velsky and Landis who first proposed it [2], can be found in the book "Fundamentals of Data Structures" by Haynes and Sasser [1]. The BITR package is a "C" written FORTRAN implementation of the AVL tree concept. In the package, each tree is a linear integer array, \( (1 - N) \), which consists of ten arrays, \( (1 - 10) \), and specifically, consists of the linked, and self-balancing, binary tree structure with \( (10) \) elements. Each node (besides array data, a leftchild (DC) and a rightchild (RC) position, and a balance factor (BF)). The tree position and the balance factor are implemented through pointers that are required for the tree traversal. All LINKAGE operations are handled internally and no external intervention in any way is required. A typical operation involves adding and updating each from the leftmost child, and also removes subprogram required for the maintenance of the structure. The time and space required for each operation is \( \Theta(\log N) \).

Typical running time

AAA, X, 00:00:00 times, respectively.

On line output of the program

The running time is very small. All runs were managed successfully. The user must set on an input oracle which is done by reading the data from an external source.

References


LONG WRITE-UP

1. Introduction

Finding elements in a list is a common programming requirement. If the list can be pregenerated, there is a simple solution. In particular, it can then be ordered prior to execution, for example by using system utilities, so that during program execution a simple binary search, which executes in $\Theta(\log n)$ times where $n$ is the number of elements in the list, can be used to find any specific element. For many applications, however, it is desirable to be able to generate the list dynamically. This implies not only the facility to search an existing list for a particular element and use the information when it is found, but, in addition, if the element is not found, to calculate the required result and add it to the list so it can be reused as necessary at a later stage in the calculation. Typically this modus operandi yields a twofold gain: needed results are calculated only once and unneeded results are never generated. Although this is clearly the procedure of choice, it is usually a nontrivial objective to achieve. The routines in the 38RTRIE package allow this to be achieved in a FORTRAN software environment.

The main difficulty associated with the dynamic generation of a list is that there is usually no way to guarantee that the labels or key, as it will be called in this article, which serves to uniquely identify list elements will come up in sequential order. This means the use of simple binary search procedures is ruled out unless the list is restructured after the addition of each new element. But that restructuring is often more costly than simply regenerating elements each time they are needed. The end result is that many users avoid the whole issue by using overkill pregeneration procedures and/or algorithms that involve a lot of redundant calculations. This is particularly true for FORTRAN applications because the language is not recursive, a feature that simplifies manipulations that are called for in dynamic generation procedures.

A height-balanced binary tree, which is also called an AVL tree after Adelson-Velsky and Landis who first proposed it in 1962, solves the problem [1]. It allows one to achieve $\Theta(\log n)$ times on search and insert operations on dynamically generated lists of length $n$ [2]. This is achieved by assigning elements to nodes in a tree and using pointers, left-child (LC) and right-child (RC), to specify, respectively, the left and right subtrees of each node. By construction the index of the left (right) subtree is less (greater) than the index of the parent node. This left-right structure applies to every node in the tree. To achieve $\Theta(\log n)$ times on fetch and insert operations on a tree with $n$ nodes it must be height-balanced. The height of any tree or subtree is defined to be the maximum number of steps from its root to its apex. Specifically, if the height of the left subtree of a node is $h_l$ and the height of the right subtree is $h_r$, then the tree is said to be height-balanced modulo one if and only if $|h_l - h_r| \leq 1$ for every node in the tree. The balance factor (BF) of a node is defined to be $h_l - h_r$. For any node in an AVL tree the balance factor is $-1, 0$ or $+1$. Examples of AVL trees are shown in fig. 1.

2. Structure of the binary tree

The tree itself is a linear integer array, $IDX \leftarrow 0$ *) The first ten elements, $IDX \leftarrow 0$ to
The key serves as a unique node identifier. It can be a single integer word (simple key) or several integer words (compound key). An allowance for compound keys is necessary because in many applications the number of required labels exceeds the number that can be packed into a single integer. The key is the complete set of integer labels. For compound keys the binary search is carried out by comparing successive key elements, with the lowest one first, the second one next, etc.

As an illustration of the use of a key to label information, consider an application that requires the evaluation of the recoupling coefficients of the SU(3) group. These recoupling coefficients are a generalization of the ordinary 6j-symbol for SU(2). The following is a commonly used notation:

\[ U\left( \lambda_1, \mu_1; \lambda_2, \mu_2; \lambda_3, \mu_3; \lambda_4, \mu_4; \lambda_5, \mu_5 \right) \]

\[ \left( \lambda_1, \mu_1 \right) \left( \lambda_2, \mu_2 \right) \left( \lambda_3, \mu_3 \right) \left( \lambda_4, \mu_4 \right) \left( \lambda_5, \mu_5 \right) \]

As shown in Fig. 3, these U-coefficients ("U" denotes Unitary) effect a transformation between the two ways of coupling three representations labeled by (1, 2, 3) to a final one labeled by (0). The total number of distinct representations involved is six, the four principal ones plus two additional ones associated with the intermediate couplings which are labeled by (12) and (34). For SU(3), each representation requires two labels, (λ, μ). The λ and μ can always be chosen to be integer and since they are always smaller than 256 for practical applications, a convenient scheme is to pack four of the labels into each of three integer words, allowing eight bits per label. The following diagram illustrates this:
FORTRAN statement function allows this to be done:

\[
\text{IPACK}(1,2,3,1) \rightarrow \text{IOR}((\text{ISHIFT}((\text{IOR}(K, \text{ISHIFT}((\text{IOR}((\text{ISHIFT}(0,0),0)),0),0)),0)),0)),0))
\]

Three integer labels that form a key that can serve as a unique identifier of the SU(3) recoupling coefficient are then,

\[
\text{LABEL}(1) = \text{IPACK}(\lambda_1, \mu_1, \lambda_2, \mu_2),\]
\[
\text{LABEL}(2) = \text{IPACK}(\lambda_2, \mu_2, \lambda_3, \mu_3),\]
\[
\text{LABEL}(3) = \text{IPACK}(\lambda_3, \mu_3, \lambda_2, \mu_2).
\]

With this scheme in place, whenever a particular SU(3) recoupling coefficient is needed, the first order of business is to generate its key (three labels) and then call for a search of the tree array where the SU(3) recoupling coefficient data is stored to see if the required coefficient has already been calculated.

**Data**

Likewise, it is necessary to allow for more than one data item. For example, a common situation is when the key refers to a range of numbers in an external array. The number of data required would then be two, one to specify the index of the first element in the array and the other to give the range.

In the SU(3) recoupling coefficient example, the \(p\) labels are running integer indices that serve to label multiplicities in the indicated couplings. For example, \(p_{12}\) distinguishes the distinct occurrences of \((\lambda_2, \mu_2)\) in the product \((\lambda_1, \mu_1) \times (\lambda_2, \mu_2)\). If \(p\) (note the bar) denotes the maximum value of \(p\), then \((p_{12} \times p_{13} \times p_{23} \times p)\) is the total number of coefficients in the set. One way to proceed would be to use the PACK function to pack the \(p\) labels into one integer word and use it as the fourth element of the key. In this case only one data item would be needed, a pointer to the position in an external array where the corresponding coefficient is stored. However, the algorithm for evaluating the SU(3) recoupling coefficients yields the complete set, all \(p\) multiplicities, in one call. It is therefore much more efficient to have the pointer point to the location in a linear array where the first element in the set can be found. If the others follow, say in speedometer order according to the \(p\) labels, then they can be easily recovered. Of course, this requires that the \(p\) values be known. But this is easily accomplished by packing them into a second data element. The individual values can always be easily recovered. For example, if \(\text{DATA}(2) = \text{IPACK}(\bar{p}_{12}, \bar{p}_{13}, \bar{p}_{23}, \bar{p})\), then the following function can be used to unpack the \(p\) labels:

\[
\text{IFRIND}(1,2) = \text{IAND}((\text{ISHEFT}(1,0),0))
\]

As a specific example,

\[
\bar{p}_{12} = \text{IFIND}(\text{DATA}(2)) = 16,\text{NBIT}(0),
\]

where \text{NBIT} is the hexadecimal number ZEF, that is, an integer with the following 13-bit structure:

\[
00000001000000000000000011111111
\]

Various scenarios for the data are possible:

(a) Simple data - each node has one data element. This could be the value of a function if it is integer. More commonly, however, it will be a pointer to an external array where the value (integer or real) can be found.

(b) Simple pointer and multiple data - each node has at least two data elements, one serving as a pointer to an external array and the other specifying the number of values to be found there. The SU(5) recoupling example is of this type.

(c) Multiple data - each node has several data elements. These could be values of a series of functions if they are integer, or pointers to external arrays where values (integer or real) are stored, or various combinations of the two.

(d) Multiple pointers and multiple data - each node has several sets of data elements pointing to different arrays and specifying the number of elements to be found in each. An illustration depicting the last scenario is given in fig. 4a.

**Long**

The number of elements dedicated to bookkeeping information in each node is fixed. For the published version of the code this number is three, one for the left-child (LC) pointer, one for the
Fig. 4. Schematic representation of (a) data and (b) link elements of a typical node in a binary tree. Like the key, data can be simple or compound. In a typical application the data points to locations in other arrays where the information (integer or real) associated with the node can be found. The left-child (LC) and right-child (RC) pointers indicate the location of the node with the next lower and higher key, respectively. The balance factor (BF) measures the difference in the heights of the left and right subtrees of the node.

right-child (RC) pointer and one for the balance factor (BF). A schematic diagram that illustrates the linkage is given in fig. 4b. Further details regarding the AVL-tree logic will not be given here as this is well-known and not the thrust of this article. Readers interested in that aspect of the problem are referred to the book by Horowitz and Sahni [2].

The theoretical maximum number of nodes a tree can have is $2^n - 1$ where $n$ is the number of bits assigned to each pointer. Since this is a large number if the pointers are full integer words, one can economize, for example, by packing the bookkeeping information into one integer word. For a tree that is height-balanced modulo one, the BF requires a minimum of two bits. On a 32-bit machine, that leaves $(32 - 2)/2 = 15$ bits each for the LC and RC pointers. Therefore such a tree could accommodate up to $2^{15} > 32767$ nodes. Since a 32k list is relatively small, this feature is not incorporated into the published version of BBTREE. Users interested in a version of the codes that uses a single integer word for the linkage information can contact the authors directly. The additional overhead associated with packing and unpacking the LC, RC, and BF tags yields approximately a 10% degradation in fetch and insert times on binary tree operations.

The actual number of nodes is simply related to the dimensioned size of the tree array. Specifically, if NK denotes the number of integer words reserved for the key and ND denotes the number of integer words set aside for data items, the length of each node is $NL = NK + ND + 1$ for the unpacked version of the codes. The $\cdot 3'$ in this expression for NL is for the BF and the LC and RC pointers. For the packed version of the codes, $NL = NK + ND + 1$. This is two less than for the unpacked version because one word rather than three is used for specifying links. The dimensioned size of the tree array must therefore be $NN = NL$ if it is to accommodate NN nodes. Actually, it must be dimensioned as $ID = 9 + NN \cdot NL$ because ten elements, $ID = 9$ to $ID(9)$, are required for specifying the structure of the tree. The function of the first ten entries is described below. So the first element of the first node is $ID(1)$, the first element of the second is $ID(NL + 1)$, ... and the last element of the nth node is $ID(n \cdot NL)$.

It is easy to see how one might exceed the dimensioned size of ID in any particular application because one rarely knows in advance the number of nodes that will be required in a calculation. For any case this number may exceed the available memory. The routines are structured in such a way that the user controls what action is to be taken if an overflow situation is encountered: stop the calculation, reinitialize the tree and continue, etc. When a tree is reinitialized the data stored in it can no longer be accessed. This introduces the possibility of a need for performing redundant calculations, the very thing one is trying to avoid by introducing the binary tree software. However, as long as the number of times a
3. Algorithms in the package

The BBTREE package consists of three subprograms: TSET, TREE and TCUT. The function of each of these programs is described in this section. A flow chart for a typical application using the BBTREE routines is given in fig. 5. It is important to note that the routines are generic and passive. They are generic because they can be used for one or more trees in an application with each tree having a different structure. And they are passive because they work on elements in the tree arrays and need no information other than that provided by the tree about itself. So, for example, there could be five fetch/insert operations on a tree ID1 and then two on a different one called ID2 followed by seven more on ID1, etc., all without resetting or reinitializing anything. Details about each of the three BBTREE routines follow.

Before an array can be used as a tree, it must be initialized. This is done with a call to TSET:

\[ \text{TSET(ID, MAXIM, NKEY, NDAT)} \]

(3.1)

**ID** = name of the linear integer array that is to be used as a tree;

**MAXIM** = maximum number of items the binary tree array will accommodate, \( \ldots \) \( 1D(-9:N) \) where \( N = MAXIM + \text{NKEY} + \text{NDAT} + 3 \);

**NKEY** = number of integer words per node dedicated to the key;

**NDAT** = number of integer words per node dedicated to data.

In addition to the obvious assignments (see the definitions for \( 1D(-9) \) to \( 1D(0) \) given in (2.1) above) a call to TSET sets \( 1D(-1) \) to zero and \( 1D(-5) \) and \( 1D(0) \), the root node pointer, to minus one. This value for \( 1D(0) \) is also a flag that serves to indicate an empty or null tree. Subsequent calls to TSET reinitialize the tree array to null status. Whenever a tree is initialized the information stored in it can no longer be accessed. Indeed, as a precautionary measure, TSET zeroes out the tree array; \( 1D(1) = 0, \ldots, 1D(N) = 0 \).

The principal routine in the package is TREE. It is used for fetch and insert operations:

\[ \text{TREE(NOADD, NEW, ID, *, *} \]  

(3.2)

**NOADD** = control parameter; if NOADD = 0 fetch only and do not try to insert a new node, see RETURN 1 & 2 below;
Fig. 5. Flow chart for a typical application using the BITREE routine. The section on the right in the dashed rectangle represents a call to the routine which has four possible outcomes: (1) RETURN 1... key already exists; (2) RETURN 2... key does not exist; but adding a new node to the tree will create an overflow condition; (3) RETURN 3... key does not exist, but NOADD = 0 an exit without attempting to add a node to the tree; and (4) RETURN (normal)... key does not exist and a new node has been added to the tree.

The first situation means the required information can be removed from storage and does not have to be recalculated. In the second case the user must decide whether to reject the entering key (YIELD) or then whether to move the tree (TMET) and continue with the calculation or simply quit. A RETURN 3 status (NOADD = 0) takes precedence over a RETURN 2 an normal RETURN. The fourth (normal RETURN) scenario is similar to what would be followed if the binary tree was not being used. Specifically, if the dashed rectangle is replaced by an arrow connecting the "start computation" and "generate data" boxes and the ends (RETURN 1, RETURN 2, RETURN 3) of the diagram discarded, one has the same program but stripped of all the BITREE technology.

NEW = key of the element being sought;
ID = name of the tree array that is to be searched;
...RETURN, executes a normal return for an unsuccessful search followed by the successful addition of a new node to the tree... (NOADD = 0);
* = RETURN 1, branch point for returning from a successful search of the binary tree, no further action required;
* → RETURN 2, branch point for returning from an unsuccessful search and a new node cannot be added to the tree without causing array overflow... (NOADD = 0);
* → RETURN 3, branch point for returning from an unsuccessful search and the addition of a node to the tree is to be suppressed... (NOADD = 0). If NOADD = 0 this entry can be suppressed in a call to TREES.

A call to TREE sets a search of ID to see if it contains a node with a key equal to NEW. If it does, the program executes a RETURN 1. If NOT is found, a node will be added to the tree provided that NOADD = 0 and the addition does not cause an overflow condition. A successful insertion yields a normal return status. In either of these cases the element ID is added to the location of the node in ID that has its key equal to NEW. If inserting a node into the tree will cause the array ID to overflow, the program executes a RETURN 2. The addition of a node to the tree is suppressed if the index NOADD = 0. In this case the program executes a RETURN 3.

A schematic of the logic used in TREE will now be given. The element being sought is X and T refers to the root node.

Step 1. Check for a null tree.
If T = nil then
  IF NOADD = 0, RETURN 3
  Else set T = X, Lchild(X) = nil, Rchild(X) = nil, and RETURN (normal)

Step 2. Compare:
Set P = T.
While P = nil do Steps 2.1 - 2.3;
Step 2.1. If Key(P) > Key(X) then P = Lchild(P)
Step 2.2. Elseif Key(P) < Key(X) then P = Rchild(P)
Step 2.3. Else RETURN 1

Step 3. Check conditions:
If NOADD = 0, RETURN 3
Else a caution will cause ID to overflow, RETURN 2

Step 4. Insert new node into the tree:
Set P = X, Lchild(X) = nil, Rchild(X) = nil
Step 5. Balance (to tree:
If height difference > 1 rotate the tree, reset the tree, and RETURN (normal).

The subroutine TOUT can be used to traverse a binary tree to find specific nodes:
TOUT(TFILE, NAD, MIN, MAX, NSTEP, ID);

TFILE = file number; index of output device
where node information is to be written,
NAD = 0, output is suppressed;
MIN = number of the first node that is to be retrieved;
MAX = number of the final node that is to be retrieved;
NSTEP = step size; nodes that are integer multiples of NSTEP beyond MIN up to
MAX will be retrieved;
ID = name of the tree array that is to be traversed.

The node number, key, and data are written out (NFILE = 0) using the following fixed format:
(IX, IK, 'Z', X, '1245', 120 (X, '1245'))

4. Performance characteristics

To test the performance of the TREE routine, a sample program was written that used a simple routine called IRANDO for generating integer random numbers between 0 and 9999 for the key and data elements of nodes in a binary tree. For testing purposes the number of integer words for the key was fixed at five and the number of data elements per node was set at two. Hence the number of integer words per node was ten (NK = 5, ND = 2, NT = NK + ND = 3 = 10). The average time required for each/insert operations using TREE on data structures of this type with between 0 and 100,000 nodes was determined using system timing routines. The results for an IBM
2000/600E and a DEC VAX-11/750 are shown in fig. 6. The time required to generate seven random numbers (G.R.N.) using IRAND on each system is also shown. The difference between the curves is a measure of the fetch/insert time per item for TREE on the IBM and DEC systems.

A similar test was made using an IBM PC/XT system that had a math coprocessor installed. In this case a memory limitation forced us to consider trees with less than 1000 nodes. The fetch/insert time required for a 1000 node tree on a PC was found to be about 13.5 ms/item. This includes the time required to generate seven random numbers, which is about 5 ms on the PC. From the results given in fig. 6 it is clear that for fetch/insert operations using the routine TREE the IBM 3090/600E runs about 75 times faster than the DEC VAX-11/750 on data sets with 30K to 40K nodes. This ratio grows with the number of nodes in the tree. The VAX system, in turn, runs between 4 and 5 times faster than the PC. Therefore, the PC executed TREE several hundred times slower than the IBM mainframe! The shapes of the curves in fig. 6 show the expected result, namely, that the fetch/insert operation execute in $O(n \log n)$ times, on both the IBM and DEC systems, where $n$ in this expression is the number of nodes in the tree.

A breakdown on the storage requirements for routines in BITREE on these three systems is given in table 1. Results are also given for DRIVER, the test run program provided with BITREE that performs fetch and insert operations on a 100 node tree of the type used in obtaining the timing information, i.e. a tree with $N = 5$ and $N D = 2$.

5. Conclusion

The routine TREE is simple to use and executes efficiently. In an application where the number of redundant calls to a subroutine is large and the time spent in the subroutine is a sizeable fraction of the total cpu time, the use of a binary tree to save information and thereby reduce redundancy can result in major gains. For example, the authors adopted this scheme in a package for calculating SU(3) $\rightarrow$ SO(3) coupling coefficients [3]. Test results showed that approximately 95% of the cpu time for large representations of SU(3) and SO(3) was spent in a subroutine called DTU3R3 of the package. Two integer words were required to tag this information. By introducing a 10000 node binary tree for storing DTU3R3 results, run times on typical applications were reduced by nearly an order of
magnitude. This illustrates the type of gains that can be realized when a binary tree scheme is implemented.

The routines in the BBTREE package have been written in a manner that makes incorporating them into existing programs an easy task. In particular, the codes are written in FORTRAN as, despite attempts to adopt simpler and more powerful languages, this is still the language of choice for most cpu-intensive scientific applications. One reason for this is the huge inventory of FORTRAN programs that is already available for the sciences. Another reason is the efficiency of FORTRAN compilers. For example, although a PASCAL version of the BBTREE package can be given that is much simpler than the FORTRAN version, due to PASCAL being a recursive language, it executes approximately three times slower than its FORTRAN counterpart on the DEC VAX-11/750 system.

In an attempt to further clarify claims made regarding ease of use for routines in the BBTREE package, on pp. 200–202 we give a skeleton listing of a subroutine called YR03 that evaluates the SU(3) recoupling coefficients referred to above. The modifications that were made to incorporate the TSET and TREE routines of the BBTREE package be set off in blocks by full lines of "///" (start) and "\///" (ending) symbols. Gains like those referred to above for DUT3R3 were realized when YR03 was substituted for XR03, a version that does not use the binary tree enhancement feature.

The purpose of the BBTREE package is to place in the public domain a set of software tools that allow scientists to incorporate database structures and logic into FORTRAN programs with a minimum of inconvenience. We have found the BBTREE routines extremely useful in numerically intensive applications. It is our hope that others will also. In this regard, the authors wish to acknowledge E.J. Reke for his early work on the project, and Drs. R.I. Henry, Y. Leachber, D.J. Mihener and G. Rosensteel for encouraging us to publish the BBTREE software package.

References


Listing

Documentation Block  Pre-occupying Coefficients for BOD

---

// Binary Tree Setup Block

Binary tree structure... labels per key: 2
- Each node: 2
  -- N=2^H elements per node
Re-initialize under the conditions:
At: 5000 (3890) nodes needed or
B: > 10000 (5000) SHARE elements

DIMENSION LABEL(32), (5000), (1:20000)
DATA IBIT, XBIT: (10000, 1:10000)

-- Packing function for binary tree labels (max 8 bits)
PACK: (1:4) (1:10, 1:16)

-- Unpacking function for extracting indices from packed array
UNPACK: (1:4) (1:10:1:16:1:16)

--- End Binary Tree Setup Block

// Binary Tree Search Block

Set indicator (ITYPE=0) assume element will be found in tree
ITYPE=0
Generate binary tree key for this case
LABEL 3 = HJACK(1AM7, 1K12, 1M2) LABEL 4 = HJACK(1AM7, 1K12, 1M2)
LABEL 5 = HJACK(1AM10, 1K12, 1M2, 1K2)

Check if first parameter (ITHEM=0)

IF ITHEMNE 0 GOTO 14

Count the number of times the binary tree is (re)initialized

(2) ITHEM*ITHEM

(Re)initialize tree

CALL TREE(1, LAM8, 1STP, 2, 2)

Search binary tree and branch if necessary as follows:

-> 812 -> overflow condition (reinitialize tree)

-> 812 -> key found. Reset the result and return

CALL TREE(1, LAM8, 1STP, 4, 12)

\S warnings

Check for overflow of SPACE and reset tree if necessary

IF ITHEM>STP GOTO 12

Proceed with the calculation. Element not found in tree (ITYPE=7)

ITYPE=7

\end Binary Tree Search

\end Information Insertion-Extraction Block

10 IDATA = PACK(IMAFF, IMAC, MINAC, MINPC)

Determine the location of the node pointer

INDEX=INDEX+1

Load pointer value and multiplicity (tree has a new node)

IF ITHEM*5 GOTO

INDEX=INDEX+1

INDEX=INDEX+1

IDATA
Pull pointer value and multiplicity (about it must pass)

E SE
      IF (GET(AREA,ILBKE) = 0)
      IF (TEST(ML, ILMTHG))
         IF (TEST(ML, ILMTHG))
            IF (TEST(ML, ILMTHG))
               ENDIF
      ENDIF

Coefficient insertion-extraction loop

COUNT=COUNT+1
DD AB=XG-2, MEGA
DD AC=XG-1, MRCB
DD AD=XG-1, MROC
DD AD=XG-1, MROC
COUNT=COUNT-1

Load the coefficients into DBNE (there has a new note)

IF (TYPE SE = 7) THEN
   DBNE(COUNT) DBNE(COUNT) DBNE(COUNT) DBNE(COUNT) DBNE(COUNT)
   E ACT
   E Continue
ENDIF

\ End Information Insertion-Extraction \\\n
====================================================================

====================================================================

END
TEST RUN INPUT

1
100
2
3
4
5

TEST RUN OUTPUT

** WARNING: ARRAY OVERFLOW ** SELECT AN OPTION
1 = PRINT THE PREVIOUS TREE AND RESET
2 = PRINT THE PREVIOUS TREE AND BUIT
3 = PRINT THE TREE
4 = JUMP MAGIC  

***** SELECTED VALUE: 100 *****
ENTRY GREATEST STEP SIZE: 
ENTRY OUTPUT FILE NUMBER:
***** SELECTED VALUE: 6000000 *****

1 2112 7257 4705 2421 1114 9660
100 978 2962 1750 9383 9700 1183 321
200 1890 7675 4468 4468 7796 3420 6728
300 2930 7741 5374 6377 1477 9278 9287
400 3970 3563 7961 5361 3000 7965 6391
500 5560 1952 9251 5433 3044 5422 8790
600 5742 2550 9294 6620 4411 4522 4306
700 6591 8307 4231 7146 8126 2929 2031
800 7829 7230 8983 6113 9110 1771 7777
900 8246 8248 7252 5317 4972 4295 9418
1000 9980 1776 7797 5087 1921 4597 3959

** PROGRAM RUN SUCCESSFULLY **

EXIT OPTIONS **

** ******************************************

VIEW THE TREE? ** SELECT AN OPTION
1 = ASCENDING ORDER
2 = DESCENDING ORDER
3 = AUTO OR STOP LOOKING

***** SELECTED VALUE: 100 *****
ENTRY START POSITION:
***** SELECTED VALUE: 20 *****
ENTRY FINAL POSITION:
***** SELECTED VALUE: 30 *****
ENTRY DESIRED STEP SIZE:
***** SELECTED VALUE: 2 *****
ENTRY OUTPUT FILE NUMBER:
***** SELECTED VALUE: 6000000 *****

20 4974 4061 4060 3930 1037 7417 6994
21 5174 3714 9991 4694 3717 5634 2060
22 4297 8941 2221 4593 8420 1911 1620
23 6500 686 947 2471 3015 7863 4920
24 7225 20a 6796 9461 9501 7545 9668
25 8667 7684 8180 5400 2664 9237 8025

S.C. Park, J.P. Draayer / balanced binary tree code 203
WITH THE NEXT NUMBER ENTER A COMMAND:

1 = ASCENDING ORDER
2 = DESCENDING ORDER
3 = QUIT OR STOP PROGRAMING

<table>
<thead>
<tr>
<th>SELECTED VALUE</th>
<th>ENTER NEXT POSITION</th>
<th>SELECTED VALUE</th>
<th>ENTER PRIOR NUMBER</th>
<th>SELECTED VALUE</th>
<th>ENTER PRIOR NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td>9</td>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

VIEW THE MODEL — SELECT AN OPTION:

1 = ASCENDING ORDER
2 = DESCENDING ORDER
3 = QUIT OR STOP PROGRAMING

SELECTED VALUE: 3