Numerical database system based on a weighted search tree

S.C. Park a,b, C. Bahri a, J.P. Draayer a,b, S.-Q. Zheng b

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

Received 7 March 1994; in final form 19 April 1994
EDITORIAL OFFICE

COMPUTER PHYSICS COMMUNICATIONS
An international journal and program library for computational physics and physical chemistry

Honorary Editor: P.G. BURKE, Belfast

Principal Editors
Geerd H.F. DIERCXSEN
Max-Planck-Institut für Astrophysik, Karl-Schwarzschild Strasse 1, D-85740 Garching bei München, Germany

James W. EASTWOOD
Culham Laboratory, Abingdon, Oxfordshire OX14 3DB, England

J.E. INGLESFIELD
ESRF, Department of physics, Catholic University of Nijmegen, Toernooiveld, 6525 ED Nijmegen, The Netherlands

F. JAMES
CERN, Data Handling Division, CH-1211 Geneva 23, Switzerland

Donald G. TRUHLAR
University of Minnesota, Department of Chemistry and Supercomputer Institute, 207 Pleasant Street S.E., Minneapolis, Minnesota 55455, USA

Program Librarian
Miss C. JACKSON
Department of Applied Mathematics, Queen's University of Belfast, Belfast BT7 1NN, N. Ireland

Program Library Director
P.G. BURKE
Department of Applied Mathematics, Queen's University of Belfast, Belfast BT7 1NN, N. Ireland

Editors in specialist fields
The complete mailing addresses of all Editors in specialist fields are printed at the beginning of each volume.

Algebraic manipulation
C.F. FISCHER, Nashville, TN
I.P. GRANT, Oxford, England
I. SHAVITT, Columbus, OH
B.T. SUTCLIFFE, Keswick, England

Astrophysics
D.G. HUMMER, Munich, Germany
M.L. NORMAN, Urbana, IL
W.M. TSCHARNUTER, Heidelberg, Germany

Atomic and molecular dynamics
B.H. BRANSDEN, Durham, England
E.A. CAREER, Los Angeles, CA
D.C. CLARY, Cambridge, England
T.H. DUNNING, Jr., Richland, WA
F.A. GIANTURO, Rome, Italy
J.-M. LAUNAY, Meudon, France
H. NAKAMURA, Nakazaki, Japan
J. READING, College Station, TX
G.C. SCHATZ, Evanston, IL
R.E. WYATT, Austin, TX

Atomic and molecular structure, spectra and properties
W. DUCH, Torun, Poland

Biophysics and biochemistry
B. MONTAGNER, PERTITT, Houston, TX

Condensed matter physics
D. FINCHAM, Keele, UK
R. HAYDOCK, Eugene, OR
O.H. NIJSELO, Lyngby, Denmark
S.B. TRICKETT, Gainesville, FL

Electromagnetics
D.A. McMAMARA, Pretoria, South Africa

High energy physics
B. van EJIK, Amsterdam, The Netherlands
C.B. LANG, Garching, Germany
J. LINNEMANN, East Lansing, MI
Y. OYANAGI, Tokyo, Japan
C. REBBI, Boston, MA

Nuclear physics
I.R. AFNAN, Adelaide, Australia
B.S. NELSSON, Copenhagen, Denmark
R. PANDEHIHRANDE, Urbana, IL

Plasma physics
D.C. BARNES, Los Alamos, NM
D.F. DUCHS, Garching bei München, Germany
D.W. HEWETT, Livermore, CA

Statistical mechanics and many body physics
K. BINDER, Mainz, Germany
M. PARRINELLO, Stuttgart, Germany

Statistics and data analysis
F. JAMES, Geneva, Switzerland
J. LINNEMANN, East Lansing, MI

Subscriptions
Computer Physics Communications (ISSN 0010-4655). For 1994, volumes 79-85 are scheduled for publication. Subscription prices are available upon request from the publisher.

Subscriptions are accepted on a prepaid basis only and are entered on a calendar year basis. Issues are sent by surface mail except to the following countries where air delivery via SAL is ensured: Argentina, Australia, Brazil, Canada, China, Hong Kong, India, Israel, Japan, Malaysia, Mexico, New Zealand, Pakistan, Singapore, South Africa, South Korea, Taiwan, Thailand, USA. For all other countries airmail rates are available upon request

Please address all your requests regarding orders and subscription queries to: Elsevier Science B.V., Journal Department, P.O. Box 211, 1000 AE Amsterdam, The Netherlands. Telephone: +31-20-5803642, Telefax: +31-20-5803598. Claims for issues not received should be made within six months of our publication (mailing) date.

Elsevier Science B.V. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the written permission of the publisher, Elsevier Science B.V., Copyright Permissions Department, P.O. Box 521, 1000 AM Amsterdam, The Netherlands.

Special regulations for authors - Upon acceptance of an article by the journal, the author(s) will be asked to transfer copyright of the article to the publisher. This transfer will vest the widest possible dissemination of information.

Special regulations for readers in the USA - This journal has been registered with the Copyright Clearance Center, Inc. Consent is given for copying of articles for personal or internal use, or for the personal use of specific clients. This consent is given on the condition that the copier pays through the Center the per-copy fee stated in the code on the first page of each article for copying beyond that permitted by Sections 107 or 108 of the US Copyright Law. The appropriate fee should be forwarded with a copy of the first page of the article to the Copyright Clearance Center, Inc., 27 Congress Street, Salem, MA 01970, USA. No code appears in an article, the author has not given broad consent to copy and permission to copy must be obtained directly from the author. The fee indicated on the first page of an article in this journal will apply retroactively to all articles published in the journal, regardless of the year of publication. This consent does not extend to other kinds of copying, such as for general distribution, resale, advertising and promotion purposes, or for creating new collective works. Special written permission must be obtained from the publisher for this copying.

US mailing notice - Computer Physics Communications (ISSN 0010-4655) is published monthly by Elsevier Science B.V., Molenwerf 1, P.O. Box 211, 1000 AE Amsterdam, The Netherlands. Annual subscription price in the USA US$ 2236.00 (valid in North, Central and South America only), including air speed delivery. Second class postage paid at Jamaica, NY 11431.

USA POSTMASTER: Send address changes to Computer Physics Communications Publications Expediting, Inc., 200 Meacham Avenue, Elmwood, NY 11003. Airfreight and mailing in the USA by Publications Expediting. No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

Published monthly
Numerical database system based on a weighted search tree

S.C. Park a,b, C. Bahri a, J.P. Draayer a,b, S.-Q. Zheng b

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

Received 7 March 1994; in final form 19 April 1994

Abstract

An on-line numerical database system, that is based on the concept of a weighted search tree and which functions like a file directory, is introduced. The system, which is designed to aid in reducing time-consuming redundant calculations in numerically intensive computations, can be used to fetch, insert and delete items from a dynamically generated list in optimal \( \sigma (\log n) \) where \( n \) is the number of items in the list] time. Items in the list are ordered according to a priority queue with the initial priority for each element set either automatically or by a user supplied algorithm. The priority queue is updated on-the-fly to reflect element hit frequency. Items can be added to a database so long as there is space to accommodate them, and when there is not, the lowest priority element(s) is removed to make room for an incoming element(s) with higher priority. The system acts passively and therefore can be applied to any number of databases, with the same or different structures, within a single application.

PROGRAM SUMMARY

Title of program: WSTREE

Catalogue number: ACTZ

CPC Program Library, Queens University of Belfast, N. Ireland (see application form in this issue)

Licencing provisions: none

Computer: IBM 3090/600J

Operating system: MVS/XA

Programming language used: FORTRAN

Memory required to execute with typical data: The seven routines that comprise the weighted search tree (wst) require a total of 25469 bytes of memory space on the IBM 3090 mainframe computer. The main program and the tree and data buffer storage requirements are over and above this base amount.

Peripherals used: none

No. of lines in distributed program, including test data, etc.: 1947 (321 in sample DRIVER, 1626 in WSTREE package)

Keywords: heap, binary tree, AVL tree, weighted search tree, linked list, multi-list representation, dynamic storage, priority strategy, numerical database, on-line database, file directory system

Nature of physical problem

Scientific computing applications frequently involve redundant calculations. This occurs either because the number of
numbers that need to be calculated is too large to be stored in memory or they occur in unknown combinations so pregeneration, which would allow them to be ordered separately and efficiently so a simple binary look up could be used, is impracticable or impossible. The \textit{wst} numerical database system introduced here [1–3] can be used to circumvent this problem as it enables one to perform the on-line search of an existing data structure for a particular element and use it if it is found, but if it is not found, it allows generated results to be added to the list so they can be reused as necessary at a later stage in the calculation. Typically this \textit{modus operandi} yields a two fold gain, namely, needed results are calculated only once and unneeded results are never generated.

When the database is full, which occurs when either the tree or associated storage arrays have reached their maximum capacity, one or perhaps even several elements, depending on the size of the incoming data set, have to be deleted before new information can be added to the database. In the sample program the delete option is exercised when the incoming element has a higher priority than the lowest priority element in the database. The deletion process checks to see if the free space generated by deleting the lowest priority node suffices to accommodate the incoming data, and if it does not additional nodes are deleted, provided or course that their priorities are less than that of the incoming node, until sufficient free space is obtained. Whenever the delete option is invoked, it is the lowest priority node that is removed from the database.

\textbf{Method of solution}

A \textit{wst} system solves the problem described above with a series of routines that have optimal time and space complexities. Within the \textit{wst}, search, insert and delete operations on dynamically generated lists of length \( n \) execute in times that go as \( \sigma(\log n) \). In addition, the \textit{wst} saves space by using a multi-list representation for the height balanced tree that is the backbone structure of the database system. A full discussion of the theory behind the \textit{wst} structure can be found in Ref. [3].

\textbf{Restrictions on the complexity of the problem}

The maximum number of nodes in a \textit{wst} is set by user. If the pointers are \( n \) bit integer words, the theoretical limit on the maximum number is \( 2^n - 1 \). In actual practice, however, the number will usually be much less than this.

\textbf{Typical running time: 0.14 ms/item on an IBM 3090/600J}

\textbf{Unusual features of the program}

Intrinsic logical AND and SHIFT functions for integers, which are called \texttt{IAND} and \texttt{ISHFT} in the \textit{wst} routines [4], are used for bit operations to encode and decode the priority and balance factor that are stored as a single integer word.

\textbf{References}


\section*{LONG WRITE-UP}

\section*{1. Introduction}

Scientific computing applications frequently involve redundant calculations. This occurs either because the number of numbers that need to be calculated is too large to store in memory or they occur in unknown combinations so pregeneration, which would allow them to be ordered efficiently so a simple binary look up could be used, is impracticable or impossible. An on-line numerical database system [1–3] can be used to circumvent this problem as it allows an existing database to be searched for a particular element and the information used when it is found, but if it is not found the element can be generated and then added to the list so it can be reused as necessary at a later stage in the calculation. Typically this \textit{modus operandi} yields a two fold gain, namely, 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 numerical database routines in the WSTREE package allow this to be achieved in a FORTRAN software environment.

The numerical database system introduced in this paper has 3 operations: search, insert and delete. An abstract data structure called a \textit{weighted search tree (wst)} [3], which is the combination of a priority queue (heap) [4] and a height-balanced AVL tree [2,3,5–7], is used to maintain the system. Each element in a list is associated with a node in a binary tree which uses left-link and right-link pointers to specify, respectively, the left and the right subtree of each node. By construction the index of the left (right)
Fig. 1. Unbalanced binary tree. The original insertion order is indicated on the left. The tree on the right is unbalanced binary tree constructed by the insertion order list on the left.

subtree is less (greater) than the index of the parent node. This left-right structure applies to every node in the tree. By requiring the tree to be height balanced, optimal $\Theta(\log n)$ time [5–7] for operations on lists of length $n$ can be achieved. The height of any tree or subtree is defined to be the maximum number of steps from its root to its apex. 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 balanced factor (BF) of a node is defined to be $h_L - h_R$. For any node in an AVL-tree the balance factor is either $-1$, $0$ or $+1$. The association of list elements with nodes in a tree is illustrated in Figs. 1 and 2 which show, respectively, the insertion order for list elements together with its unbalanced and height-balanced representations.

Fig. 2. Height-balanced binary tree. The number above the circles are the balanced factors, $-1$ if the subtree to the right is one node longer than the one to the left, $0$ if the left and right subtrees are the same length, and $+1$ if the subtree to the left is one node longer than the one to the right.
Fig. 3. Multi-list representation. (a) Simple typical type multilist representation. (b) Schematic representation of (a). Each node for both representations has two links which allow one to compute not only left child and right child but also parent node position.

The data structure that is used for constructing our numerical database system is actually a modification of a height-balanced binary tree. As indicated above, a normal height-balanced binary tree has two links per node which point to the left child and right child of the parent node. The new data structure, which is a multi-list representation [3,7,8] of height-balanced tree, also has two links per node. However, these two links allow one to determine not only the left child and right child but the position of the parent node as well. This feature is important because maintaining the wst heap structure, which is required for superimposing on the tree an organization of the nodes according to their priority, requires parent node position information. A portion of a multi-list representation of a tree is shown in Fig. 3. The feature that two links in multi-list representation suffices to additionally compute the position of the parent node saves space and renders the system space optimal. Fig. 4 is a multi-list representation of the height balanced binary tree shown in Fig. 2.

Our on-line database system has a delete operation. This is an essential feature when dealing with data structures that are either larger than the memory capacity of the machine being used or one is working in a programming environment that does not allow for dynamic allocation of storage. When the database is full, which occurs when either the tree or associated storage arrays have reached their maximum capacity, one or perhaps even several elements, depending on the size of incoming data set, have to be deleted before the new information can be stored in the database. Of course, the deletion option should only be exercised if the incoming element has a higher priority than the lowest priority element in the database. The deletion process checks to see if the free space generated by deleting the
lowest priority node suffices to accommodate the incoming data, and if it does not, additional nodes are deleted, provided of course that their priorities are less than that of the incoming node, until sufficient free space is obtained. Whenever the deletion option is invoked, it is always the lowest priority node that is deleted from the database. The ordering of node in the tree according to priority is maintained by the heap structure, see Fig. 4b. Specifically, the ordering is kept such that the priority of the \( i \)th node is always lower than that of the \( 2 \times i \)th and \( 2 \times (i + 1) \)th nodes where \( 1 \leq i \leq \lfloor n/2 \rfloor \) and \( n \) is the maximum number of nodes in the tree [4]. The first node always has the lowest priority and is therefore the one that is deleted when the deletion option is exercised.

The database system introduced here, functions like a management system for disk files. Specifically, the system handles the bookkeeping for storing data (files) of various lengths in (on) a fixed size array (disk). Data are stored in the array as linked allocations [9] with the \( \ast wst \) (file directory algorithm) managing the space. As shown in Fig. 5, the location of information associated with a node in the \( \ast wst \) is determined by two data elements stored with each node, a data location or head pointer, which is an index that points to the location in a BUFFER array where the first element is actually stored and, as well, the location in the associated link-list array called LLBUFF that holds the pointer for the second element, etc., and a counter size, which specifies the number of fixed size blocks in storage associated with the node. For example, in Fig. 5, the head pointer is 4 and the size of the data set is 6. The 4 is the index of BUFFER where the first element is found as well as the index of LLBUFF which specifies the location of the next element, LLBUFF(4) = 5, and so on. Hence, as indicated, the 6 data elements associated with the node are located in BUFFER(4), BUFFER(5), BUFFER(6), BUFFER(7),...
Fig. 5. Data allocation. The nodes in a WST can be used as a directory system. The two data entries LO and SZ that follow the key indicate the location and size of a data associated with the node. In the example shown LO = 4 and SZ = 6. The elements in LLBUFF point to blocks in BUFFER that are associated with the node. Given LO and SZ, one can retrieve all the data in BUFFER by using LLBUFF. The crosshatched blocks in BUFFER are occupied by data elements.

BUFFER(10), and BUFFER(11). Note that there is a redundancy that is built in as LLBUFF(11) = -1 indicates that there are no additional elements associated with the node. This redundancy is necessary to achieve efficient management of the system since if the size of a block is not known, element by element checking would be necessary to determine, for example, the amount of free space that could be gained by deleting it from the database.

For applications that have a single or fixed number of elements associated with each node, the redundancy referred to above can be removed without any loss as only the first of the two data elements is needed to maintain the structure of the database. In fact, for the fixed-buffer size scenario the link-list array is unnecessary as only fixed-length blocks of data are stored in the buffer and only the starting address or location is required for retrieving any specific one. As this can lead to considerable savings, it is handled as a separate special case in the WST package. A further simplification can be realized if the stored information is fixed-length integer data as it can then be stored directly in the tree as additional data elements per node, a situation that even eliminates the need for keeping a data location index. As the latter is a very trivial modification of the fixed-buffer size scenario it is not treated separately in the WST package. In the discussion that follows, the general case is presented with comments regarding special cases only given when the simplifications to the general results are not straightforward. For the management of free space, the free blocks in the buffer are linked together as shown in Fig. 6, with a link-list specification using the first three entries of LLBUFF in the same way as for the storage of data.
Data

Information can be stored directly in the tree as integer data, provided the number of such items per node is fixed. Two data elements are reserved for specifying the location and size of information stored in the auxiliary buffer array. This suffices because the data allocation type used for the auxiliary buffer is a linked list so specifying the head of the list and the total number of elements in it is sufficient to retrieved all members. Specifically, if the auxiliary array is called BUFFER and I is the head pointer stored in the WST as the node location element, then the first data element is BUFFER(I) and the second and all subsequent elements are also found in BUFFER(I) but with the simple substitution I ← LLBUFF(I) where LLBUFF is an integer array that maintains the linked list. The number of reserved data elements can be reduced to one and the need for a link-list array eliminated when the information stored in the buffer has a fixed-length rather than a mixed length. Therefore the WST package treats the fixed-buffer and mixed-buffer length cases separately, see below, since the former can be handled with less overhead and therefore with algorithms that execute more efficiently.

Priority and balance factor

The priority and balance factor are packed into a single integer word. Since the balance factor (BF) requires only two bits, the first two left-most bits in an integer word are used for it and the other 30 bits are used for the priority. The priority scheme consists of two parts: a base priority which reflects the initial value of the node and is under user control and the hit frequency which measures the frequency of use of the node information. The scheme employed in the sample program is a simple one: the first 8 right-most bits are reserved for the base priority and the remaining 22 bits for the hit-frequency times the base priority. The actual value assigned for the priority is therefore the hit frequency times the base priority plus the base priority. For example, if a particular data time has a base priority of 7 and a current hit frequency of 16 then the current assigned priority value is simply $16 \times 7 \times 256 + 7 = 28679$. To decode the priority and balance factor information the intrinsic functions ISHIFT and IAND can be used [10]. For example, IAND(I,Z3FFFFFFF) can be used on an IBM system to extract the priority from I and ISHIFT(I, −30) gives the balance factor. In the WST a balance factor of 0 is used to indicate that the heights of left-subtree and right-subtree are equal, while if it is 2 the height of the left-subtree is one greater than that of the right-subtree, and if it is 3 the height of left-subtree is one less than that of the right-subtree. To encode the priority IP and balance factor IB into a single integer I, the elementary function ISHIFT(IB,30) is added to IP, $I = ISHIFT(IB,30) + IP$.

Links

The number of elements dedicated to bookkeeping information in each node is fixed. For the published version of the code this number is two, one for the left-link (LL) pointer and one for the right-link (RL) pointer. A schematic diagram that illustrates the linkage is given in Fig. 3. Further details regarding the WST logic will not be given here. Readers interested in that aspect of the problem are referred to Ref. [3]. 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.

The first eleven elements of the tree array, ID(−10) to ID(0), are parameters that specify its structure and provide other information associated with the fetch, insert and delete operations:

- ID(−10) = number of nodes currently in the tree;
- ID(−9) = maximum nodes in the tree;
- ID(−8) = location of the parent node at ID(−7);
- ID(−7) = location of the node to be balanced;
ID(-6) = location of the parent node at ID(-5);
ID(-5) = location of the current node;
ID(-4) = number of integer words per node for the key;
ID(-3) = number of integer words per node for the key and data;
ID(-2) = ID(-3) + 1, location of the priority and balance factor in a node;
ID(-1) = location of the next available node in the tree array;
ID(0) = root node pointer (negative one for a null tree).

As pointed out above, the number of integer data elements per node is at least two, one to specify the location of the first element in the auxiliary buffer array and the other to give its size. (Integer data can be stored directly in the tree – eliminating the need for an auxiliary buffer – provided the number of elements per node is fixed.)

3. Algorithms in the package

The WSTREE package consists of seven main subprograms: TSET, TCHK, TADD, TINS, TDEL, TOUT, TMRG. In typical applications, calls to only five (TSET, TCHK, TADD, TOUT, TMRG) of these seven subroutines are made. However, TADD uses two additional subroutines, TINS and TDEL, for inserting and deleting nodes from the WST. The function of each of these programs is described in this section. A flow chart for a typical application using the WSTREE routines is given in Fig. 7. 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 seven WSTREE routines follows.

Before an array can be used as a tree, it must be initialized. This is done with a call to the subprogram TSET which has two entries, one for the tree with an auxiliary linked-list [TSETLL] and another for the one without (list-free) [TSETLF]:

TSET ...

entry TSETLL(ID,MXNODE,NKEY,NDAT,LLBUFF,MXBUFF)
entry TSETLF(ID,MXNODE,NKEY,NDAT)

ID = linear integer array that is to be used as a tree, ID(-10:MXSIZE) where MXSIZE = (MXNODE + 1)*(NKEY + NDAT + 3);
MXNODE = maximum number of nodes the WST array will accommodate;
NKEY = number of integer words per node dedicated to the key;
NDAT = number of integer words per node dedicated to the data;
LLBUFF = linear integer array that holds pointer information for the buffer, LLBUFF(-2:MXBUFF);
MXBUFF = size of the linear buffer array where information is stored, that is BUFFER(1:MXBUFF).

In addition to the obvious assignments, see the definitions for ID(-10) to ID(0) given above, a call to TSET initializes the tree by setting ID(-10), ID(-5) and ID(-1) to zero and ID(-8), ID(-7), ID(-6) and ID(0), the root node pointer, to minus one. This value for ID(0) is also a flag that serves to indicate
Fig. 7. Flow chart for a typical application using the WSTREE routines. After a KEY is generated, TCHK is called to see if the key already exists in the database. If the answer is yes, the priority is increased and data retrieved. If the answer is no, the insertion procedure is continued. Before an item can be inserted, the tree and free space $fs$ in the buffer must be checked to see if both are sufficient for the incoming data $cs$. If the answer is yes, the incoming record will be inserted (TINS). If the answer is no, items will be deleted from the database, provided their priorities $lp$ are lower than that of the incoming record $cp$, until there is sufficient space to accommodate the new record.

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 zeros out the tree array: $ID(1) = 0$, ..., $ID(N) = 0$.

As explained above, the integer array LLBUFF holds the pointer information for locations in the array BUFFER where the information associated with each node is stored. This link-list array is
dimensioned LLBUFF(−2:MXBUFF) with LLBUFF(1) to LLBUFF(MXBUFF) used for pointers to actual data elements in BUFFER and LLBUFF(−2) to LLBUFF(0) used for information on the free space in BUFFER. Specifically, LLBUFF(−2) is the head of free space, LLBUFF(−1) is the tail of free space, and LLBUFF(0) is the size of free space. A call to TSET sets LLBUFF(−2) to 1, and LLBUFF(−1) and LLBUFF(0) to MXBUFF.

Before describing the subprograms TCHK, TADD, TINS and TDEL, the algorithms for computing the position of the left child and right child of a particular node need to be introduced. To make the WSTREE package execute efficiently these algorithms have not been separated out as subroutines in the WSTREE package, nonetheless, for convenience they are identified here as if they are subroutines called LCHILD and RCHILD, respectively. The return value $p$ in LCHILD and RCHILD are the left child position and right child position, respectively.

LCHILD($p$)
if LLINK($p$) = nil then
  $p \leftarrow$ nil
else if RLINK(LLINK($p$)) = $p$ then
  if BF($p$) = −1 then
    $p \leftarrow$ LLINK($p$)
  else
    $p \leftarrow$ nil
else
  $p \leftarrow$ LLINK($p$)

RCHILD($p$)
if LLINK($p$) = nil then
  $p \leftarrow$ nil
else if RLINK(LLINK($p$)) = $p$ then
  if BF($p$) = −1 then
    $p \leftarrow$ nil
  else
    $p \leftarrow$ LLINK($p$)
else
  $p \leftarrow$ RLINK(LLINK($p$))

The subprogram TCHK is used for fetch operations. A call to TCHK effects a search of ID to see if it contains a node with a key equal to NKEY. If it does, the program executes a RETURN 1 after increasing the priority of the node that was found by an amount equal to 256 times the base priority and reconstructing the heap. If NKEY is not found, the program executes a normal RETURN and ID(−5) is assigned information on where to insert NKEY into the wst when added via a call to TINS. A schematic of the logic used in TCHK will now be given. The element being sought is $x$ and $t$ refers to the root node:

TCHK(NKEY,ID,*)

NKEY = linear integer array containing the key of the incoming item;
ID = linear integer array that is being used for the tree.
Step 1. Check for a null tree:
   if \( t = \text{nil} \) then
       goto Step 3

Step 2. Compare:
   set \( p \leftarrow t \)
   while \( p \neq \text{nil} \) do Steps 2.1–2.3:
      Step 2.1 if \( \text{KEY}(p) > \text{KEY}(x) \) then \( t \leftarrow p \); call LCHILD(p)
      Step 2.2 elseif \( \text{KEY}(p) < \text{KEY}(x) \) then \( t \leftarrow p \), call RCHILD(p)
      Step 2.3 else
          increase the priority of node \( t \);
          reconstruct the heap;
      RETURN 1
   endwhile

Step 3. RETURN (normal)

The subroutine TADD is used to add an element to an existing database provided space is available. If space is not available and the priority of the incoming element is greater than the lowest priority element currently in the database that element is eliminated to make room for the new one. The last scenario is repeated as necessary and appropriate in an attempt to accommodate the new element. During the execution of the TADD procedure, calls to the subroutines TDEL for deleting the lowest priority node and TINS for inserting the incoming node in the tree are made. These two subroutines are described below. A call to TADD can only be done if the linked-list LLBUFF has been initialized in TSETLL. The logic of the subroutine TADD is the following:

TADD(NKEY,NDAT,BULOAD,NOSIZE,NPBASE,ID,BUFFER,LLBUFF)

NKEY = linear integer array containing the key of the incoming item;
NDAT = linear integer array containing the data of the incoming item;
BULOAD = linear real array were the incoming data elements are stored;
NOSIZE = size of the incoming data item;
NPBASE = base priority that is to be assigned to the incoming item;
ID = linear integer array that is being used for the tree;
BUFFER = linear array where the BULOAD elements are to be stored;
LLBUFF = linear integer array that hold pointer information for BUFFER.

Step 1. Check if either the tree or buffer is full:
   if yes, check if the priority of the incoming item is higher than that of lowest priority node in the tree
      if yes, delete lowest priority node
          (call TDEL)
          goto Step 1
      else goto Step 3
Step 2. Insert the incoming item into the database (call TINS)
Step 3. RETURN
The subroutine TINS is used to insert a node into the wst. TINS must be preceded by a call to TCHK, which checks to see whether or not the item NKEY is already in the wst. After inserting NKEY into the wst, TINS reconstructs the heap.

TINS(NKEY,ID)

NKEY = linear integer array containing the key of the incoming item;
ID = linear integer array that is being used for the tree.

Step 1. Insert new node into the tree:
   if tree is empty then
      Set p ← x, left child of x ← nil, right child of y ← nil
Step 2. Balance the tree:
   If height difference > 1 rotate the tree, reset the tree
Step 3. Reconstruct the heap
Step 4. RETURN (normal)

The subroutine TDEL deletes the lowest priority node from a wst. This is a necessary feature for an
general fixed-space, on-line system since a situation can be encountered where either the tree or the auxiliary
buffer is full, and a node with a higher priority than the lowest priority node in the tree needs to be
added to the wst. The subroutine TDEL can be executed to make space for the incoming item.

TDEL(ID)

ID = linear integer array that is being used for the tree.

Step 1. Find x along a path from the root node to node y:
   p ← ROOT;
   while (p ≠ y) do
      if (p satisfies one of the conditions for x) then x ← p;
      if (KEY(p) > KEY(y)) then call LCHILD(p);
      else if (KEY(p) < KEY(y)) then call RCHILD(p);
   endwhile
Step 2. Delete node y in the linear tree array:
   if (left child of y = null) then
      if (right child of y ≠ null) then call RCHILD(y)
      else y ← null
   else if (right child of y = null) then call LCHILD(y)
   else begin
      z ← the node contains the largest key value smaller than KEY(y);
      i ← y;
      while (i ≠ null) do
         if (i satisfies one of the conditions for x) then x ← i;
         if (KEY(i) > !KEY(z)) then call LCHILD(i);
         else if (KEY(i) < KEY(z)) then call RCHILD(i);
      endwhile
      y ← z;
   end
The subroutine TINS is used to insert a node into the wst. TINS must be preceded by a call to TCHK which checks to see whether or not the item NKEY is already in the wst. After inserting NKEY into the wst, TINS reconstructs the heap.

TINS(NKEY,ID)

NKEY = linear integer array containing the key of the incoming item;
ID = linear integer array that is being used for the tree.

Step 1. Insert new node into the tree:
   if tree is empty then
      Set p ← x, left child of x ← nil, right child of y ← nil
Step 2. Balance the tree:
   If height difference > 1 rotate the tree, reset the tree
Step 3. Reconstruct the heap
Step 4. RETURN (normal)

The subroutine TDEL deletes the lowest priority node from a wst. This is a necessary feature for a fixed-space, on-line system since a situation can be encountered where either the tree or the auxiliary buffer is full, and a node with a higher priority than the lowest priority node in the tree needs to be added to the wst. The subroutine TDEL can be executed to make space for the incoming item.

TDEL(ID)

ID = linear integer array that is being used for the tree.

Step 1. Find x along a path from the root node to node y:
   p ← ROOT;
   while (p ≠ y) do
      if (p satisfies one of the conditions for x) then x ← p;
      if (KEY(p) > KEY(y)) then call LCHILD(p);
      else if (KEY(p) < KEY(y)) then call RCHILD(p);
   endwhile
Step 2. Delete node y in the linear tree array:
   if (left child of y = null) then
      if (right child of y ≠ null) then call RCHILD(y)
      else y ← null
   else if (right child of y = null) then call LCHILD(y)
   else begin
      z ← the node contains the largest key value smaller than KEY(y);
      i ← y;
      while (i ≠ null) do
         if (i satisfies one of the conditions for x) then x ← i;
         if (KEY(i) > (KEY(z))) then call LCHILD(i);
         else if (KEY(i) < KEY(z)) then call RCHILD(i);
      endwhile
      y ← z;
   end
Step 3. Balance the tree:

\[
\text{while } (x \text{ is not a leaf}) \text{ do} \\
\quad \text{if (height difference of subtrees of } x > 1) \text{ then rotate;} \\
\quad \text{if (KEY}(y) < \text{KEY}(x)) \text{ then call RCHILD}(x); \\
\quad \text{else call LCHILD}(x); \\
\text{endwhile}
\]

Step 4. Reconstruct the heap
Step 5. RETURN (normal)

The subroutine TOUT can be used to traverse \textit{wst} to find specific nodes:

\[
\text{TOUT(NFILE,NAD,MINT,MINT,NSTEP,ID)}
\]

\text{NFILE} = file number of the output device where node information is to be written; if \text{NFILE} = 0 output is suppressed;

\text{NAD} = 1 if the traversal is to be in traversed in ascending order, otherwise it will be traversed in descending order;

\text{MIN} = number of the first node that is to be retrieved;

\text{MAX} = number of the final node that is to be retrieved;

\text{NSTEP} = step size; nodes that are integer multiples of \text{NSTEP} beyond \text{MIN} up to \text{MAX} will be retrieved;

\text{ID} = name of the tree array that is to be traversed.

The node number, key, and data are written out (\text{NFILE} \neq 0) using the following fixed format: (1X, I6, ',', 2X, 14I5/((10X,14I5)).

The subroutine TMRG can be used to merge two trees that have the same structure. The logic of the routine is very similar to that employed in TOUT. It is included in the WSTREE package so datasets that are generated independently can be merged into a final one. This means a dataset can be generated either as time becomes available on a particular machine or by independent processors running in parallel. By using TMRG, a collection of independently generated datasets can be merged, two at a time, into a final data structure.

\[
\text{TMRG(ID1,BUFF1,LLBF1,ID2,BUFF2,LLBF2,*)}
\]

\text{ID1} = linear integer array 1 that is being used for the tree;

\text{BUFF1} = linear array 1 where the data elements are to be stored;

\text{LLBF1} = linear integer array 1 that hold pointer information for \text{BUFF1}.

\text{ID2} = linear integer array 2 that is being used for the tree;

\text{BUFF2} = linear array 2 where the data elements are to be stored;

\text{LLBF2} = linear integer array 2 that hold pointer information for \text{BUFF2}.

Elements from the first tree are merged into the second. Duplicate entries are eliminated and each entry maintains its assigned priority. If the merge results in an overflow condition, notification is given and the lowest priority item is eliminated in favor of an incoming element with higher priority. If the two trees
WSTREE package are time-optimal, the disk input-output is the slow part of this procedure. But this too can be avoided if rather than writing the output from TOUT to disk it is used directly by TINS to load another duplicate database. Though faster, this procedure would requires double the space.

A breakdown on the storage requirements for routines in WSTREE on an IBM 3090/600J is given in Table 1.

6. Conclusion

The routines in the WSTREE package are simple to use and execute efficiently. In applications where the number of redundant calls to a subprogram is large and the time spent in that subroutine is a sizable fraction of the total cpu time, the use of a wrs for saving intermediate results and thereby reducing the redundancy can result in major gains. An important feature of the WSTREE database system is that while in many ways it is like a file directory system, it goes beyond this because nodes can be assigned priorities and when the assigned storage space is used up there is a deletion mechanism available that allows the user to keep the most valuable information in the database at the expense of throwing out the least valuable.

The routines in the WSTREE 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
Table 1
Storage requirements in bytes for routines in the WSTREE package on an IBM 3090/600J. Numbers in parenthesis include array storage for 25000 nodes in the tree and a buffer and pointer arrays of 50000 words each.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Storage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSET</td>
<td>916</td>
</tr>
<tr>
<td>TCHK</td>
<td>3060</td>
</tr>
<tr>
<td>TADD</td>
<td>1132</td>
</tr>
<tr>
<td>TINS</td>
<td>4940</td>
</tr>
<tr>
<td>TDELT</td>
<td>7944</td>
</tr>
<tr>
<td>TOUT</td>
<td>1876</td>
</tr>
<tr>
<td>TMRG</td>
<td>5600</td>
</tr>
<tr>
<td>TOTALS(WSTREE)</td>
<td>25468</td>
</tr>
<tr>
<td>IRAND</td>
<td>444</td>
</tr>
<tr>
<td>ISIZE</td>
<td>444</td>
</tr>
<tr>
<td>DRIVER</td>
<td>2804</td>
</tr>
</tbody>
</table>

(1402956)

<table>
<thead>
<tr>
<th>TOTALS(ALL)</th>
<th>Storage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>29160</td>
</tr>
</tbody>
</table>

(1429312)

Attempts to adopt simpler and more powerful languages, this is still the language of choice for most cpu-intense scientific applications. One reason for this is the ready availability of many FORTRAN codes that have been tuned for efficiency, which is particularly important in high-performance computing applications, for scientific applications. Another reason is the efficiency of FORTRAN compilers is generally very high.

The purpose of the WSTREE 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 WSTREE routines extremely useful in numerically intensive applications. It is our hope that others will also.

References