Proceedings of the 2007 Western Multiconference on Computer Simulation: International Conference on High Level Simulation Languages and Applications, January 14-17, 2007, San Diego, California, pp. 150-155

Braided AVL Trees for Efficient Event Sets and Ranked Sets in the SIMSCRIPT III Simulation Programming Language

Stephen V. Rice
The University of Mississippi
P.O. Box 1848, University, MS 38677 USA
rice@cs.olemiss.edu

Keywords: data structures and algorithms for discrete-event simulation, event set management

Abstract

The doubly-linked list used to store event sets and ranked sets in SIMSCRIPT discrete-event simulation programs is replaced by a "braided AVL tree," which is a single data structure that is both a doubly-linked list and an AVL tree. Using this data structure, event scheduling and ranked set insertions are efficiently performed in logarithmic time.

1 INTRODUCTION

SIMSCRIPT III is an object-oriented programming language for discrete-event simulation [1]. It is a recent major extension of the SIMSCRIPT II.5 language. A SIMSCRIPT III "set" is an ordered collection of "members," which may be objects or entity records. The ordering may be specified as FIFO (first-in first-out), LIFO (last-in first-out), or ranked based on the values of one or more attributes (fields) of the members. Each set has an "owner," which may be an object, class, or entity. For example, the following SIMSCRIPT III statements declare that `Job` objects may be members of a set named `Queue` which is owned by a `Machine` object. The members are stored in the set in order of decreasing value of their `Priority` attribute.

```
begin class Job
   every Job
      has a Priority and belongs to a Queue
   define Priority as an integer variable
   define Queue as a set
      ranked by high Priority
end

begin class Machine
   every Machine owns a Job'Queue
end
```

A "file" statement places a member into a set, and a "remove" statement removes a member from a set. The following example illustrates the creation of a `Job` object; assignment of 10 as its `Priority`; filing of the object into a set owned by a `Machine` object; and subsequent removal of the object from the set.

```
create Job
Priority(Job) = 10
file Job in Job'Queue(Machine)
remove Job from Job'Queue(Machine)
```

In SIMSCRIPT II.5, a set is implemented as a doubly-linked list. The owner of the set has implicitly-defined attributes containing pointers to the first and last members of the set, or containing null pointers (zeros) if the set is empty. Each member has implicitly-defined attributes containing pointers to its predecessor and successor in the set. The links are not circular: the predecessor is null for the first member and the successor is null for the last member.

Having a linked-list data type built into the language is greatly convenient for programming simulation models. The program may traverse the linked list, accessing the members one by one, either in the forward direction (starting with the first member and following the successor links) or in reverse (starting with the last member and following the predecessor links). A loop that traverses a linked list is expressed elegantly in SIMSCRIPT III:

```
for each Job in Job'Queue(Machine)
do
   …
loop
```

In the SIMSCRIPT II.5 implementation of sets, filing a member into a FIFO or LIFO set, and removing a member from a set, are performed efficiently in *O(1)* time, i.e., constant time. However, filing a member into a ranked set of *n* members is performed by a linear scan of the set to locate the correct position for the new member, which requires *O(n)* time.

SIMSCRIPT III retains the SIMSCRIPT II.5 implementation of FIFO and LIFO sets. However, a ranked set is stored in a data structure that is both a doubly-linked list and an AVL tree. The linked list allows the set to be traversed as before, while the AVL tree permits fast insertions. The time required to file a member into a ranked set is thereby reduced from *O(n)* to *O(lg n)*. The time required to remove a member from a ranked set, however, increases from *O(1)* to *O(lg n)*.

Ranked sets are used to implement the future-event set in SIMSCRIPT II.5 and SIMSCRIPT III. Future events are ordered by increasing time of occurrence. Events are scheduled more rapidly in SIMSCRIPT III because of the use of AVL trees for fast insertion into the future-event set.

An AVL tree is a height-balanced binary search tree with the property that for each node of the tree, the height of the node's left subtree differs from the height of the node's right subtree by at most one. The height of an unbalanced binary tree of *n* nodes is *O(n)* in the worst case, whereas the height of an AVL tree is at most *1.44 lg n* or *O(lg n)*.

The AVL tree data structure is named for Russian mathematicians Adel'son-Vel'skiĭ and Landis who invented it in 1962 [2]. It is described in many books on algorithms. The author of this paper surveyed a dozen such sources seeking pseudocode for an AVL tree implementation. Where pseudocode could be found, it was incomplete, usually lacking the "delete" operation. Moreover, none incorporated a doubly-linked list into the data structure. The author therefore developed the required pseudocode and presents it in this paper. The pseudocode makes minimal use of recursion for the sake of efficiency, and the "delete" operation is simplified by direct access to the deleted object's predecessor. The author has written the logic in the SIMSCRIPT III language and validated its correctness by thorough testing. Performance results of the SIMSCRIPT III version are presented at the conclusion of this paper.

Although linked lists and AVL trees are well known, the combined data structure described in this paper is not well known. A data structure that is simultaneously a linked list and a binary search tree has been called a "braided binary search tree" by Laszlo [3] and a "linked list tree" by Pfaff [4]. We prefer Laszlo's name and refer to our data structure as a "braided AVL tree." Note that this data structure is not the same as a "threaded binary tree," which stores a direct pointer to a node's predecessor (successor) only when the node has no left (right) child in the tree. It differs also from Henriksen's linked list indexed by a separate binary search tree [5]. Henriksen's data structure has been used for efficient event sets in implementations of the GPSS, SLX, and SLAM simulation programming languages [6]. Our braided AVL trees are used for efficient event sets and efficient ranked sets in the SIMSCRIPT III programming language.

2 IMPLEMENTATION

In addition to pointers to the first and last members, each owner of a ranked set has an implicitly-defined pointer to the member that is the root of the AVL tree (which is null if the set is empty). Each member of a ranked set has pointers to its predecessor and successor, and four more implicitly-defined attributes:

- *balance*, which is either –1, 0, or +1, equal to the height of the right subtree minus the height of the left subtree; if balance is –1, we say the tree rooted at this member is "left heavy"; if balance is +1, the tree is "right heavy";
- *left*, which points to the left child (or is null if there is no left child);
- *right*, which points to the right child (or is null if there is no right child);
- *parent*, which points to the parent (or is null if there is no parent, i.e., the member is the root of the entire tree).

Figure 1 illustrates an AVL tree. Beneath each node is the node's balance factor, either –1, 0, or +1. Figure 2 shows the same AVL tree together with the predecessor and successor links of the doubly-linked list.

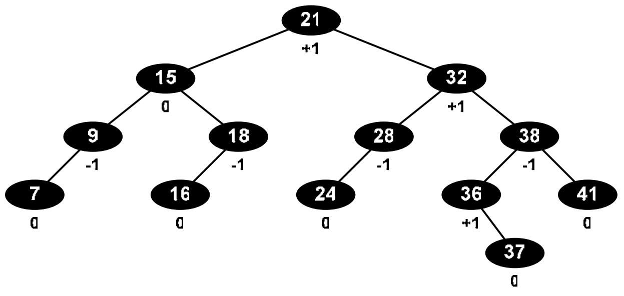

Figure 1. An AVL tree showing balance factors

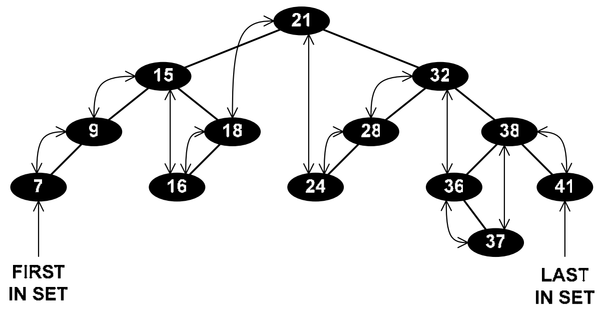

Figure 2. A braided AVL tree with predecessor and successor links

Inserting or deleting a node from an AVL tree may cause a balance factor to become –2 or +2. In this case, it is necessary to perform a rotation. There are four types of rotations: single right, double right, single left, and double left. These are illustrated in Figures 3, 4, 5, and 6, respectively.

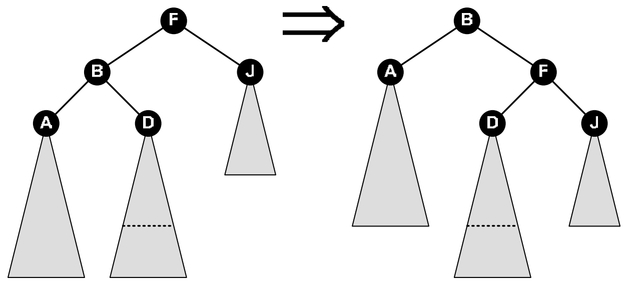

Figure 3. Single right rotation

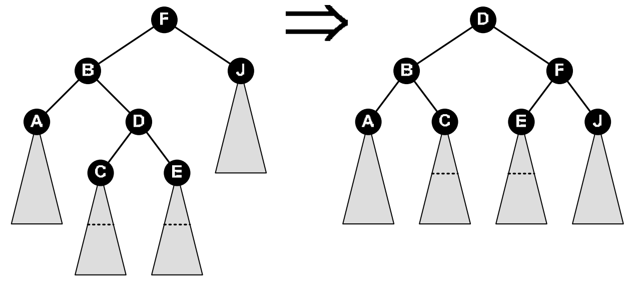

Figure 4. Double right rotation

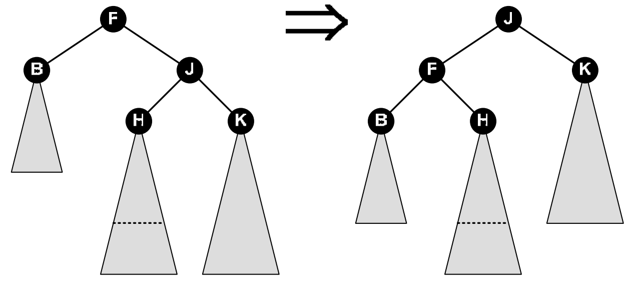

Figure 5. Single left rotation

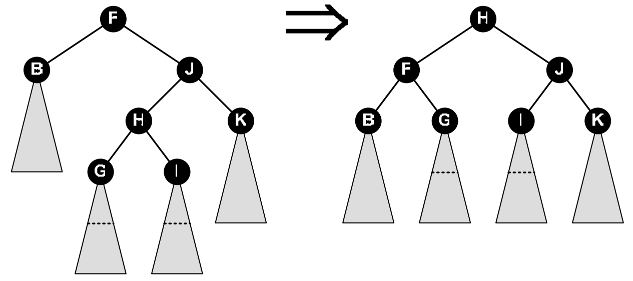

Figure 6. Double left rotation

The shaded triangle beneath node A in Figure 3 represents a subtree of height *h*. The triangle beneath node D has a horizontal dashed line to represent a subtree of height *h* or *h–1*. The height of the subtree rooted at node J is *h–1*. The left subtree of node F has height *h+1* and the right subtree of node F has height *h–1*; therefore, node F has a balance factor of –2. After a single right rotation, node F has a balance factor of –1 or 0, and node B becomes the new root. A single rotation suffices if node B has a balance factor of –1 or 0 before the rotation. However, if node B is right heavy, as shown in Figure 4, then a double rotation is required. (One or both of nodes C and E have height *h* in Figure 4.)

Each rotation is performed in *O(1)* time. All four types of rotations are handled by the *Rotate* function in Figure 7.

The *Adjust* function, shown in Figure 8, determines when rotations are required after a member has been inserted or deleted. The process begins with the parent of the inserted or deleted member, and proceeds up toward the root of the tree. At most one rotation is needed for an insertion, but a rotation may be required at each level of the tree for a deletion. The number of levels is *O(lg n)* and the work required at each level is performed in constant time; hence, the running time of the *Adjust* function is *O(lg n)*.

The *Insert* function, shown in Figure 9, starts from the root of the tree and travels downward to locate the position to insert the new member. The number of steps is limited by the height of the tree and so the insertion occurs in *O(lg n)* time. The *Adjust* function is then called to perform a rotation if required. It also takes *O(lg n)* time; therefore, the running time of the *Insert* function is *O(lg n)*. Using a technique called "scripting," the SIMSCRIPT III compiler generates the code that compares the values of the ranking attributes.

The *Delete* function, shown in Figure 10, handles two cases. In Case 1, the member to be deleted has zero or one child. The child, if any, moves up in the tree to take the place of the deleted member. The *Adjust* function is then called to perform rotations as needed. Case 1 takes *O(lg n)* time.

In Case 2, the member to be deleted has two children. However, the predecessor (and also the successor) of the member is guaranteed to have zero or one child. We recursively call the *Delete* function to delete the predecessor using Case 1, which takes *O(lg n)* time. Then in constant time, we put the predecessor back into the tree in place of the member to be deleted. The overall running time of the *Delete* function is *O(lg n)*. Note that having a direct pointer to the predecessor, as provided by the doubly-linked list, facilitates the deletion.

```
function Rotate(root)

    // This function rotates the tree rooted at "root"
    // and returns the new root of the tree.

    bal ← balance(root);
    if bal = –1  // left heavy, need to rotate right
        newroot ← left(root)
    else  // bal = +1; right heavy, need to rotate left
        newroot ← right(root)
    end if

    if balance(newroot) = – bal  // double rotation is required
        p ← newroot;
        if bal = –1  // double right rotation
            newroot ← right(p);  q ← left(newroot);
            right(p) ← q;  left(newroot) ← p
        else  // double left rotation
            newroot ← left(p);  q ← right(newroot);
            left(p) ← q;  right(newroot) ← p
        end if
        if q ≠ nil  parent(q) ← p  end if
        parent(p) ← newroot;
        if balance(newroot) = bal
            balance(root) ← – bal
        else
            balance(root) ← 0
        end if
        if balance(newroot) = – bal
            balance(p) ← bal
        else
            balance(p) ← 0
        end if
        balance(newroot) ← 0
    else  // only a single rotation
        balance(newroot) ← balance(newroot) – bal;
        balance(root) ← – balance(newroot)
    end if

    if bal = –1
        p ← right(newroot);  left(root) ← p;  right(newroot) ← root
    else
        p ← left(newroot);  right(root) ← p;  left(newroot) ← root
    end if

    if p ≠ nil  parent(p) ← root  end if
    parent(newroot) ← parent(root);
    parent(root) ← newroot;

    p ← parent(newroot);
    if p ≠ nil
        if left(p) = root
            left(p) ← newroot
        else
            right(p) ← newroot
        end if
    end if

    return newroot

end function
```

Figure 7. *Rotate* function

```
function Adjust(root, aroot, side, change)

    // This function adjusts the tree rooted at "root" when the
    // height of the left (side = –1) or right (side = +1) subtree of
    // "aroot" has shrunk (change = –1) or grown (change = +1),
    // and returns the new root of the tree.

    done ← false;

    repeat
        if balance(aroot) = (side × change)  // rotation is required
            if aroot = root
                return Rotate(aroot)
            end if
            aroot ← Rotate(aroot);
            if change = +1 or balance(aroot) ≠ 0
                done ← true
            end if
        else
            if balance(aroot) = 0
                balance(aroot) ← (side × change);
                if change = –1
                    done ← true
                end if
            else
                balance(aroot) ← 0;
                if change = +1
                    done ← true
                end if
            end if
        end if
        if not done
            p ← parent(aroot);
            if p = nil
                done ← true
            else
                if left(p) = aroot
                    side ← –1
                else
                    side ← +1
                end if
                aroot ← p
            end if
        end if
    until done

    return root

end function
```

Figure 8. *Adjust* function

```
function Insert(root, member)

    // This function inserts "member" into the tree rooted at "root"
    // and returns the new root of the tree.

    balance(member) ← 0;
    left(member) ← nil;
    right(member) ← nil;

    if root = nil  // the set is empty
        parent(member) ← nil;
        predecessor(member) ← nil;
        successor(member) ← nil;
        return member
    end if

    // find where "member" belongs
    aroot ← root;
    side ← 0;
    pred ← nil;
    succ ← nil;
    repeat
        if member sorts ahead of aroot  // scripted comparison
            succ ← aroot;
            if left(aroot) = nil
                side ← –1;
                left(aroot) ← member
            else
                aroot ← left(aroot)
            end if
        else  // member sorts after aroot
            pred ← aroot;
            if right(aroot) = nil
                side ← +1;
                right(aroot) ← member
            else
                aroot ← right(aroot)
            end if
        end if
    until side ≠ 0

    parent(member) ← aroot;
    predecessor(member) ← pred;
    if pred ≠ nil  successor(pred) ← member  end if
    successor(member) ← succ;
    if succ ≠ nil  predecessor(succ) ← member  end if

    return Adjust(root, aroot, side, +1)

end function
```

Figure 9. *Insert* function

```
function Delete(root, member)

    // This function deletes "member" from the tree rooted at "root"
    // and returns the new root of the tree.

    if left(member) = nil or right(member) = nil  // Case 1
        child ← left(member);
        if right(member) ≠ nil
            child ← right(member)
        end if
        aroot ← parent(member);
        if aroot = nil
            if child ≠ nil
                parent(child) ← nil;
                predecessor(child) ← nil;
                successor(child) ← nil
            end if
            return child
        end if
        pred ← predecessor(member);
        succ ← successor(member);
        if pred ≠ nil  successor(pred) ← succ  end if
        if succ ≠ nil  predecessor(succ) ← pred  end if
        if child ≠ nil  parent(child) ← aroot  end if
        if left(aroot) = member
            left(aroot) ← child;  side ← –1
        else
            right(aroot) ← child;  side ← +1
        end if
        return Adjust(root, aroot, side, –1)
    end if

    // Case 2
    p ← predecessor(member);
    newroot ← Delete(root, p);  // delete the predecessor of "member"

    // the predecessor now takes the place of "member"

    balance(p) ← balance(member);
    left(p) ← left(member);
    if left(p) ≠ nil  parent(left(p)) ← p  end if
    right(p) ← right(member);
    if right(p) ≠ nil  parent(right(p)) ← p  end if

    parent(p) ← parent(member);
    if parent(p) = nil
        newroot ← p
    else
        if left(parent(p)) = member
            left(parent(p)) ← p
        else
            right(parent(p)) ← p
        end if
    end if

    predecessor(p) ← predecessor(member);
    if predecessor(p) ≠ nil  successor(predecessor(p)) ← p  end if
    successor(p) ← successor(member);
    if successor(p) ≠ nil  predecessor(successor(p)) ← p  end if

    return newroot

end function
```

Figure 10. *Delete* function

Lastly, Figure 11 shows how the *Insert* function is called for a SIMSCRIPT III "file" statement, and Figure 12 shows how the *Delete* function is called for a SIMSCRIPT III "remove" statement.

```
procedure File(owner, member)

    // This procedure places "member" in
    // the ranked set owned by "owner."

    root(owner) ← Insert(root(owner), member);

    if predecessor(member) = nil
        first(owner) ← member
    end if

    if successor(member) = nil
        last(owner) ← member
    end if

end procedure
```

Figure 11. *File* procedure

```
procedure Remove(owner, member)

    // This procedure removes "member" from
    // the ranked set owned by "owner."

    root(owner) ← Delete(root(owner), member);

    if first(owner) = member
        first(owner) ← successor(member)
    end if

    if last(owner) = member
        last(owner) ← predecessor(member)
    end if

end procedure
```

Figure 12. *Remove* procedure

Table 1. Time required to insert *n* members into a SIMSCRIPT III ranked set

n	*AVL Tree*	*Linear Scan*
10,000	0.02 seconds	0.59 seconds
100,000	0.24 seconds	409.69 seconds
1,000,000	3.27 seconds	would take hours
10,000,000	45.20 seconds	would take days

3 CONCLUSION

The AVL tree algorithm inserts *n* members into a ranked set in *O(n lg n)* time, whereas a linear scan requires $O(n^2)$ time for *n* insertions. Table 1 shows for four values of *n*, the time required to file *n* members with randomly-selected rank values into an empty set. These times were determined using the SIMSCRIPT III implementation of the algorithms running on a PC with a 2.4 GHz Intel Pentium 4 processor.

Without the AVL tree, it takes *O(n)* time to delete *n* members from a set. With the AVL tree, it takes *O(n lg n)* time to delete *n* members because of the need to rebalance the tree after each deletion. However, the effect on running time is negligible compared to the savings gained on insertion. The time needed to remove 100,000 members from a set is 0.09 seconds without the AVL tree and 0.10 seconds with the AVL tree.

In future work, we will measure the speed improvements obtained when running "typical" simulation models written in SIMSCRIPT III. We may also utilize the AVL tree to speed up ranked set searches. The following loop can be executed in *O(lg n)* time using the AVL tree versus *O(n)* time for a linear scan.

```
for each Job in Job'Queue(Machine)
with Priority(Job) = P
   find the first case
```

REFERENCES

[1] Rice, S.V.; A. Marjanski; H.M. Markowitz; S.M. Bailey. 2005. "The SIMSCRIPT III Programming Language for Modular Object-Oriented Simulation." In *Proceedings of the 2005 Winter Simulation Conference* (Orlando, FL, Dec. 4-7). IEEE, Piscataway, NJ, 621-630.

[2] Adel'son-Vel'skiĭ, G.M. and E.M. Landis. 1962. "An Algorithm for the Organization of Information." *Soviet Mathematics Doklady 3*, 1259-1263.

[3] Laszlo, M.J. 1996. *Computational Geometry and Computer Graphics in C++*. Prentice Hall, Upper Saddle River, NJ.

[4] Pfaff, B. 2004. "Performance Analysis of BSTs in System Software." In *Proceedings of SIGMETRICS 2004/Performance 2004: The Joint International Conference on Measurement and Modeling of Computer Systems* (New York, June 12-16). ACM Press, New York, 410-411.

[5] Henriksen, J.O. 1983. "Event List Management—A Tutorial." In *Proceedings of the 1983 Winter Simulation Conference* (Arlington, VA, Dec. 12-14). IEEE, Piscataway, NJ, 543-551.

[6] Leemis, L.M. and S.K. Park. 2006. *Discrete-Event Simulation: A First Course*. Pearson Prentice Hall, Upper Saddle River, NJ.