ACL2-Certified AVL Trees

Ryan Ralston
School of Computer Science University of Oklahoma
200 Felgar Street
Norman, Oklahoma
strawdog@ou.edu

ABSTRACT

AVL trees supply operations for information storage and retrieval in time proportional to the logarithm of the number of items stored and in a space increment, beyond the space needed for the information itself, that increases linearly with the number of items. The public operations of insertion, deletion, and retrieval are supported internally by rotations to maintain order and balance properties that make it possible to meet logarithmic performance requirements. This report describes an AVL implementation in ACL2. It focuses on verifying correctness issues of order, balance, and conservation of keys.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

AVL tree formalization in ACL2

Keywords

software engineering, AVL trees, ACL2

1. AVL DATA STRUCTURE

AVL trees provide one of many ways to maintain information associated with unique, ordered keys in such a way that insertion, deletion, and retrieval can be carried out in time proportional to the logarithm of the number of items, with a space increment (beyond what is required to store the information and the keys) proportional to the number of items. To accomplish these goals the algorithm performs a standard set of tree rotations as needed during insertion and deletion to maintain at all nodes in a binary tree (1) an order among the keys that places smaller keys in left subtrees and larger ones on the right and (2) a balance property that requires the left and right subtrees of each node to have comparable heights.

The implementation in this paper defines a tree structure with four fields: key, left subtree, right subtree, and height, with field selectors named key, lf, rt, and ht, respectively. All basic operations incrementally update the height field and correctness of these updates is verified in the formal model.

```
(defun ht=ht-meas? (tr)
  (or (empty? tr)
      (and (= (ht tr) (ht-meas tr))
           (ht=ht-meas? (lf tr))
           (ht=ht-meas? (rt tr)))))

(defun balanced? (tr)
  (or (null tr)
      (and (non-empty? tr)
           (<= (abs (- (ht-meas (lf tr))
                       (ht-meas (rt tr))))
               1)
           (balanced? (lf tr))
           (balanced? (rt tr)))))
```

Figure 1: Height and balance

Operations on AVL trees must preserve order and balance, and this implementation uses the intrinsic lexorder to compare keys, which is a total ordering for ACL2 objects. For compactness, the presentation uses "=<" for lexorder and "∼" for not equal.

```
(defun ordered? (tr)
  (or (empty? tr)
      (and (ordered? (lf tr))
           (ordered? (rt tr))
           (or (empty? (lf tr))
               (let* ((k (tree-max (lf tr))))
                 (and (=< k (key tr))
                      (~= k (key tr)))))
           (or (empty? (rt tr))
               (let* ((k (tree-min (rt tr))))
                 (and (=< (key tr) k)
                      (~= k (key tr))))))))
```

Figure 2: Ordered trees

Figure 2 displays the predicate for detecting properly ordered trees. It refers to functions that compute the minimum and maximum key in a tree. Those functions are defined conventionally.

2. ROTATIONS

The rotations needed to maintain balance can be defined as non-recursive operations on the tree structure. Figure 3 displays the code for basic right rotation.

```
(defun easy-R (tr)
  (let* ((R (avl (key tr)
                 (rt (lf tr))
                 (rt tr)
                 (ht-incr (rt (lf tr)) (rt tr))))
         (L (lf (lf tr)))
         (H (ht-incr L R)))
    (avl (key (lf tr)) L R H)))
```

Figure 3: Basic right rotation

```
(defthm easy-R-preserves-order
  (implies (and (easy-R-able? tr)
                (ordered? tr))
           (ordered? (easy-R tr)))
    :hints
  (("Goal"
    :use ((:instance tree-min<=-tree-max
                     (tr (rt tr))))
    :in-theory (disable tree-min<=-tree-max))))
```

Figure 4: Basic right rotation preserves order

A basic rotation must preserve the required ordering among the keys. That is, the rotated tree must be properly ordered if the original tree was. The theorem of Figure 4 confirms that property for right rotation. A hint tells ACL2 to use a theorem relating tree-min and tree-max specifically on the right subtree, but to avoid using it in other parts of the proof. ACL2's rewrite stack fills up with a loop repeatedly rewriting with the theorem if it is not disabled. However, the right tree needs help rewriting the tree comparisons on right rotations, so the specific instance is forced with the use-instance hint.

```
(defun left-heavy? (tr)
  (if (non-empty? tr)
      (= (ht-meas (lf tr))
         (+ (ht-meas (rt tr)) 2))
      nil))

(defun hard-R (tr)
  (let* ((L (easy-L (lf tr)))
         (R (rt tr))
         (H (ht-incr L R)))
    (easy-R (avl (key tr) L R H))))

(defun rot-R (tr)
  (let* ((L (lf tr))
         (htL (ht (lf L)))
         (htR (ht (rt L))))
    (if (< htL htR)
        (hard-R tr)
        (easy-R tr))))

(defun rebal-R (tr)
  (let* ((ht-L (ht (lf tr)))
         (ht-R (ht (rt tr))))
    (if (= ht-L (+ ht-R 2))
        (rot-R tr)
        tr)))
```

Figure 5: Right rotation

When an insertion or deletion operation delivers a subtree that is out of balance, one or two basic rotations bring it back into balance, depending on whether the most out-of-balance portion is on the outside or inside subtree of the taller portion of the unbalanced tree. Figure 5 displays this operation.

```
(defthm rot-R-restores-bal-left
  (implies (and (ht=ht-meas? tr)
                (left-heavy? tr)
                (balanced? (lf tr))
                (balanced? (rt tr)))
           (balanced? (lf (rot-R tr)))))

(defthm rot-R-restores-balance
  (implies (and (ht=ht-meas? tr)
                (left-heavy? tr)
                (balanced? (lf tr))
                (balanced? (rt tr)))
           (balanced? (rot-R tr)))
  :hints
  (("Goal"
    :use ((:instance rot-R-restores-bal-left)
          (:instance rot-R-restores-bal-right)))))
```

Figure 6: Restoring balance

```
(defthm rot-R-preserves-keys
  (implies (and (or (easy-R-able? tr)
                    (hard-R-able? tr))
                (ht=ht-meas? tr)
                (in-tree? k tr))
           (in-tree? k (rot-R tr)))
  :hints
  (( "Goal"
     :hands-off (easy-L easy-R hard-L hard-R))))

(defthm rot-R-conserves-keys
  (implies (and (or (easy-R-able? tr)
                    (hard-R-able? tr))
                (ht=ht-meas? tr)
                (not (in-tree? k tr)))
           (not (in-tree? k (rot-R tr))))
  :hints
  (( "Goal"
     :hands-off (easy-L easy-R hard-L hard-R))))
```

Figure 7: Rotation conserves keys

A rotation is applied only when a tree is out of balance in a specific way. Figure 5 displays a predicate that detects this specific, out-of-balance situation for rotations to the right[1]. To be sure that the implementation is correct, ACL2 must confirm that, under the conditions that call for rotation to the right, the rotation brings the tree back into balance. ACL2 is able to prove this if it is directed to make use of lemmas that confirm the balance property in the subtrees of the rotated tree. The rebalance theorem appears in Figure 6, along with one of the lemmas.

3. CONSERVATION OF KEYS

In addition to preserving balance and order, operations on AVL trees must conserve keys. That is, the keys in a tree

[1]The terminology "heavy" to describe an unbalanced node replaces the standard concept of tree height with weight to be more consistent with the phrase "balancing".

produced by an operation must be exactly the same keys that resided in the original tree, plus one more key in the case of the insertion, and one less in the case of the deletion.

The first stage in proving that insertion (Figure 8) and deletion (Figure 11) conserve keys is to verify that the rotations do so. As usual, hints guide the proof engine along a productive path through the maze of potential inferences. Figure 7 provides a few of the details.

```
(defun ins (k tr)
  (cond ((empty? tr) (avl k nil nil 1))
        ((== k (key tr)) (avl k
                              (lf tr)
                              (rt tr)
                              (ht tr)))
        ((=< k (key tr))(let*((x (key tr))
                              (L (ins k (lf tr)))
                              (R (rt tr))
                              (H (ht-incr L R)))
                         (rebal-R (avl x L R H))))
        (t              (let*((x (key tr))
                              (L (lf tr))
                              (R (ins k (rt tr)))
                              (H (ht-incr L R)))
                         (rebal-L (avl x L R H))))))
```

Figure 8: Insertion

One way to describe key conservation for insertion is to split it into three theorems (Figure 9). The first states that the only new key after insertion is the key that is being inserted. The second says that any key in the tree before insertion is still in the tree afterwards, and the third says that any key that is not in the original tree and is not the inserted key is not in the tree after insertion. These theorems make their way through the ACL2 logic, facilitated by conservation lemmas for rotations (Figure 7) [2].

4. PRESERVATION OF ORDER

The next property to verify is preservation of order under insertion and deletion. If a key is inserted or deleted from an empty or ordered tree, the resulting tree is also ordered.

There are at least two ways to approach the issue of order and the choice between them informs the definition of the order predicate. One approach is to observe that all keys to the left of a node are less than the node's key value. Similarly, the keys to the right of a node are greater than the node's value. For this approach, the order predicate uses a function that checks every node in the left and right subtrees against the node's key.

Another approach is to compare the node's key to the maximum key value in the left subtree and the minimum key value in the right subtree.

The second approach separates the concepts. The tree-max

[2]A single iff combining the two implications would be equivalent, logically, to the preservation and conservation theorems of Figure 9. However, ACL2 does not succeed in proving the combined theorem. Probably, the theorems can be successfully combined into the more compact "iff" form, but it would require tactics not yet discovered in the present work.

```
(defthm insert-key-is-in-tree
  (implies (ht=ht-meas? tr)
           (in-tree? k (insert k tr)))
  :hints (( "Goal" :hands-off (rebal ht-incr))))

(defthm insert-preserves-keys
  (implies (and (ht=ht-meas? tr)
                (or (in-tree? k tr)
                    (== k j)))
           (in-tree? k (insert j tr)))
  :hints (( "Goal" :hands-off (rebal ht-incr))))

(defthm insert-conserves-keys
  (implies (and (ht=ht-meas? tr)
                (not (in-tree? k tr))
                (~= k j))
           (not (in-tree? k (insert j tr))))
  :hints
  (( "Goal"
     :hands-off (rebal ht-incr)
     :use ((:instance ht-meas=ht-when-ht=1)))))
```

Figure 9: Insert conserves keys

```
(defthm insert-tree-max-lemma
  (implies (and (ht=ht-meas? tr)
                (non-empty? tr)
                (ordered? tr)
                (~= k (key tr))
                (=< k (key tr)))
           (=< (tree-max (ins k (lf tr)))
               (key tr))))

(defthm insert-ordered-is-ordered
  (implies (and (ht=ht-meas? tr)
                (ordered? tr))
           (ordered? (ins k tr)))
  :hints (("Goal" :hands-off
                  (rebal-L rebal-R ht-incr))))
```

Figure 10: Ordering after insertion

and tree-min functions only take the tree as a parameter so it is much easier to prove lemmas about those concepts. Figure 10 displays one of the tree-max lemmas and the insert-preserves-order theorem.

Figure 11 defines the functions related to deletion. Commonly, the chain of descendents on the right of a node are considered the tree's "spine". The function moves the deepest node on the spine (referred to as the "sacrum") to the current location in the tree. The shrink function returns the sacrum's left subtree. Figure 12 shows preservation of keys for deletion.

5. RELATED WORK

Gamboa and Cowles developed an implementation of red-black trees with many correctness properties verified, plus a detailed performance analysis [1]. The present work makes use of some of the Gamboa/Cowles proof strategies to push forward proofs in the AVL formalization.

Proofs of properties concerning preservation of order among keys, for example, need many detailed lemmas about the comparator. In a few cases, the needed lemmas trend towards the arcane. For example, ACL2 proof attempts re-

```
(defun shrink (tr)
  (cond ((empty? tr) (list *any-key* tr))
        ((empty? (rt tr)) (list (key tr) (lf tr)))
        (t (let* ((k-tr (shrink (rt tr)))
                  (k (car k-tr))
                  (L (lf tr))
                  (R (cadr k-tr))
                  (H (ht-incr L R))
                  (shrunken (avl (key tr) L R H)))
             (list k (rebal-R shrunken))))))

(defun raise-sacrum (tr)
   (let* ((k-tr (shrink (lf tr)))
          (k (car k-tr))
          (L (cadr k-tr))
          (R (rt tr))
          (H (ht-incr L R)))
     (rebal-L (avl k L R H))))

(defun del (k tr)
  (cond
    ((empty? tr) tr)
    ((== k (key tr))(if (empty?(lf tr))
                        (rt tr)
                        (raise-sacrum tr)))
    ((=< k (key tr))(let* ((L (del k (lf tr)))
                           (R (rt tr))
                           (H (ht-incr L R)))
                      (rebal-L(avl (key tr) L R H))))
    (t (let*((L (lf tr))
             (R (del k (rt tr)))
             (H (ht-incr L R)))
         (rebal-R(avl (key tr) L R H))))))
```

Figure 11: Deletion

vealed a connection between the transitive and anti-reflexive properties, namely that an increasing sequence of four keys in which the last key in the sequence is the same as the first implies that the keys are equal. The red-black tree implementation relied on such a lemma, which bolstered confidence that the lemma could be helpful in the AVL proofs.

Nipkow and Pusch have developed a partial implementation of AVL trees with properties verified by the Isabelle proof assistant [2]. Their approach is to first build an inefficient implementation that does not record heights incrementally in the tree. They carry out the initial proofs in that domain, then build an efficient implementation with recorded heights and verify that the height records in the efficient implementation match heights measured in the inefficient implementation.

Both the Gamboa/Cowles red-black implementation and the Nipkow/Pusch AVL implementation focus on insertion. Deletion is included in the library discussed in this paper, although verification of balance after deletion is not yet complete.

6. CONCLUSION AND FUTURE WORK

This report has presented a formalization of AVL trees with mechanical verification of certain aspects of preservation of order, balance and conservation of keys for insertion and deletion. Currently, balance is not fully verified for deletion and the verification of balance on insertion assumes insertion increases the height by at most 1. We plan to address these issues in future work.

```
(defthm shrink-preserves-keys-not-key
  (implies (and (ht=ht-meas? tr)
                (non-empty? tr)
                (in-tree? k tr)
                (not (equal k (car (shrink tr)))))
           (in-tree? k (cadr (shrink tr))))
  :hints (("Goal" :hands-off (rebal-R ht-incr))))

(defthm shrink-preserves-keys-key
  (implies (and (ht=ht-meas? tr)
                (non-empty? tr)
                (in-tree? k tr)
                (not (in-tree? k (cadr (shrink tr)))))
           (equal (car (shrink tr)) k))
  :hints (("Goal" :hands-off (rebal-R ht-incr))))

(defthm delete-preserves-keys
  (implies (and (ht=ht-meas? tr)
                (in-tree? k tr)
                (not (equal k j)))
           (in-tree? k (del j tr)))
  :hints (("Goal" :hands-off (rebal-L rebal-R ht-incr))))
```

Figure 12: Preservation on deletion

7. REFERENCES

[1] R. Gamboa and J. Cowles. Implementing a cost-aware evaluator for ACL2 expressions. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 71–80, New York, NY, USA, 2006. ACM.

[2] T. Nipkow and C. Pusch. AVL Trees, http://afp.sourceforge.net/entries/AVL-Trees.shtml, 2004.