Verification of Insertion Sort

ACL2 Lecture 2

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

Computer Science Department
University of Texas
2317 Speedway, M/S D9500
Austin, TX 78712-0233

May, 2021

Verification of Insertion Sort

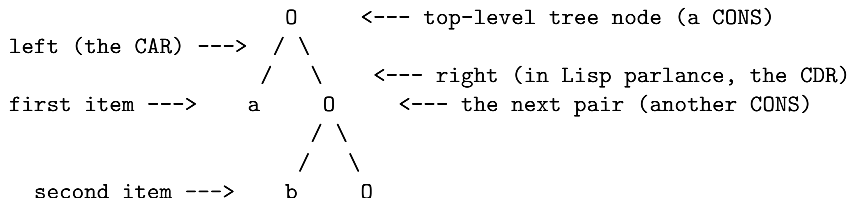
In this lecture, we investigate what it means to have an ordered list of elements, and we present a method for ordering such a list.

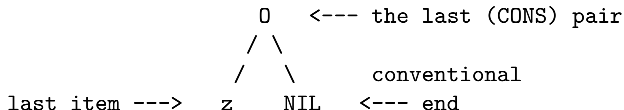
- We define an ACL2 predicate that recognizes when a list is sorted.
- We specify an insertion-sort algorithm.
- We explore what it means for an algorithm to produce a sorted result.
- We use ACL2 to prove the correctness of an insertion-sort algorithm.

At the end, we provide some challenge problems.

Aggregating a Collection of ACL2 Objects

We will use lists to create collections of ACL2 objects.





where each item (a, b, ..., z) are the elements.

Recognizing Ordered Lists

We recognize an ordered list with help from ACL2's built-in comparison function LEXORDER.

```
(defun orderedp (x)
  (if (atom x)
      t
    (if (atom (cdr x))
        t
      (and (lexorder (car x) (cadr x))
           (orderedp (cdr x))))))
```

If this Boolean function returns T, then we say that its input is ordered.

Note, in running text, we write ACL2 terms in upper case because Lisp *up-cases* everything.

Note also, that ORDEREDP does allow duplicate entries.

Insertion

Insertion sort on lists involves inserting elements into an already sorted list.

```
(defun insert (e x)
  (if (atom x)
      ;; Create single-item list
      (list e)
    (if (lexorder e (car x))
        ;; If "less than or equal", insert E at the front
        (cons e x)
      ;; Otherwise, make (CAR X) the first item
      (cons (car x)
            ;; and insert E in the rest of X
            (insert e (cdr x))))))
```

For the function `INSERT` to work properly, the list `X` must be `ORDEREDP`.

INSERT Each Item into an Ordered List

The function ISORT (below), repeatedly (recursively) inserts its first element, in the sorted remainder of the list.

Empty and single-element lists are already sorted.

ISORT uses INSERT (above) to place one item into an already-sorted list.

```
(defun isort (x)
  (if (atom x)
      nil
    ;; Insert the first element of X,
    (insert (car x)
            ;; into the sort of the rest of X
            (isort (cdr x)))))
```

This process is repeated until all items (working from the end of the list to the front) have been inserted.

Examples of Insertion and Sorting

`INSERT`ion into an ordered list preserves order.

```
(insert 3 '(1 2 4 5))  ==>  (1 2 3 4 5)
```

`INSERT`ion into an unordered list places the item in front of the first larger element.

```
(insert 3 '(1 4 2 5))  ==>  (1 3 4 2 5)
```

```
(isort '(1 3 4 2 5))  ==>  (1 2 3 4 5)
```

```
(isort '(a c d b e))  ==>  (A B C D E)
```

The ACL2 `LEXORDER` function can order any ACL2 objects.

```
(isort '(a 1 #\c '(f) "d" -3/2))  ==>  (-3/2 1 #\c "d" A '(F))
```

Insertion into a Sorted List

The INSERT function places item E where it is *less than or equal* to the first element in X.

If the X is ordered, then INSERT will place E in a manner that will leave the extended list ordered.

```
(defthm orderedp-insert
  ;; Given ordered list X, INSERT produces an ordered result
  (implies (orderedp x)
           (orderedp (insert e x))))
```

This is proved by induction on the right-associated, tree-based, structure of X.

- **Base Case:** For (ATOM X), prove that a one-element list is ordered.
- **Induction:** For (CONSP X), given ordered X, then inserting E produces a sorted extension.

ISORT Produces an ORDEREDP Result

Given any list of items, ISORT will order it.

```
(defthm orderedp-isort
  ;;  ISORT returns ordered result
  (orderedp (isort x)))
```

The function LEXORDER is able to order any two ACL2 objects.

LEXORDER is used by INSERT to place one element into an ORDEREDP list.

From the end of the list to the front, ISORT repeatedly used INSERT to place one item in an evolving ORDEREDP list.

Is Being ORDEREDP Sufficient?

Consider the SORT1 function:

```
(defun sort1 (x)
  (declare (ignorable x))
  '(a b c))
(defthm SORT1-appears-to-be-OK
  (equal (sort1 '(c b a)) '(a b c)))
(defthm orderedp-sort1
  (orderedp (sort1 x)))
```

It appears that SORT1 has similar properties as ISORT, but:

```
(defthm sort1-is-not-OK
  (equal (sort1 '(f g h))   ;; Arragh!!!
         '(a b c)))
```

ISORT Should Maintain the Number of Objects

It's obvious that ISORT should return as many objects as it receives.

```
(defun len (x)
  (if (atom x) 0 (+ 1 (len (cdr x)))))
```

```
(defthm len-isort
  (equal (len (isort x))
         (len x)))
```

That seems better... but consider the SORT0 function.

```
(defun sort0 (x)
  (if (atom x)
      nil
    (cons 0 (sort0 (cdr x)))))
(defthm orderedp-sort0
  (orderedp (sort0 x)))
```

```
(defthm len-sort0
  (equal (len (sort0 x))
         (len x)))
```

SORT0 seems to yield a good result, but its first item is always 0!

Sorting Should Return the Same Elements

In addition to the result being ORDEREDP, we wish to specify that sorting returns the same number of every object.

```
(defun how-many (e x)
  (if (atom x)
      0
    (if (equal e (car x))
        (1+ (how-many e (cdr x)))
      (how-many e (cdr x)))))
```

Given any `E`, this function counts how many times it appears in list `X`.

For all items, we want the number of items to be the same in the sorted result.

ISORT Returns the Same Number of Each Different Item

Do our INSERT and ISORT functions maintain the number of items?

We can prove:

```
(defthm how-many-insert
  ;; INSERT increased number of ``i'' elements
  (equal (how-many e (insert i x))
         (if (equal e i)
             (1+ (how-many e x))
           (how-many e x))))
(defthm how-many-isort
  ;; Number of each element unchanged
  (equal (how-many e (isort x))
         (how-many e x)))
```

We now know that ISORT returns the same collection of objects.

Something to Consider: In-Place Sorting

Imagine a memory where we access and update individual memory locations.

Can we write a *memory-based* sorting function? And prove it correct?

We want a memory model accessed by `NTH` and updated by `UPDATE-NTH` — our memory look-up and update operators.

```
+-----+--    ---+-----+-----+--    ---+-----+--    ---+-----+
|  E  |  o o o  |  a  |  b  |  o o o  |  z  |  o o o  |  W  |
+-----+--    ---+-----+-----+--    ---+-----+--    ---+-----+
0               L                           R               end
   Our memory     <-- Sort from L up to R -->
```

And, we want to define and prove the correctness of an *in-memory* insertion sort procedure for a sub-sequence (from L up to R) of memory values.

Can you define such a in-memory sorting algorithm using ACL2's `NTH` and `UPDATE-NTH` functions?

And, can you write a function `(M-TO-L m l r)` that projects the elements from memory, `m`, starting at position `l` up to (but not including) position `r`?

Extra Problems

```
(defun mem (a x)
  (if (atom x)
      nil
    (or (equal a (car x))
        (mem a (cdr x)))))
```

```
(defun rm (e x)
  (if (atom x)
      nil
    (if (equal e (car x))
        (cdr x)
      (cons (car x)
            (rm e (cdr x))))))
```

```
(defun perm (x y)
  (if (atom x)
      (atom y)
    (and (mem (car x) y)
         (perm (cdr x)
               (rm (car x) y)))))
```

Given the definitions above, prove:

```
(defthm perm-reflexive
  (perm x x))
```

```
(defthm perm-symmetric
  (implies (perm x y)
           (perm y x)))
```

```
(defthm perm-transitive
  (implies (and (perm x y)
                (perm y z))
           (perm x z)))
```