Laboratory: Insertion Sort

Summary: In this lab, we explore a variety of issues related to the insertion sort algorithm.

Contents:

- **Exercises**
  - Exercise 0: Preparation
  - Exercise 1: Testing Insert
  - Exercise 2: Inserting Strings
  - Exercise 3: Generalizing Insertion
  - Exercise 4: Displaying Steps in Insertion Sort
  - Exercise 5: Checking Potential Problems
  - Exercise 6: Generalizing Insertion Sort
  - Exercise 7: Observing insert!

Exercises

Exercise 0: Preparation

a. Start DrScheme.

b. Copy the code from the accompanying reading into DrScheme.

Exercise 1: Testing Insert

a. Test both versions of the `insert-number` procedure from the reading by inserting the number 42
   - into an empty list;
   - into a list of numbers larger than 42, arranged in ascending order;
   - into a list of numbers smaller than 42, arranged in ascending order;
   - into a list of numbers both smaller and larger than 42, arranged in ascending order; and
   - into a list that contains only three copies of 42 (that is, the list created by `(list 42 42 42)`).

b. What happens if the list is *not* in ascending order when `insert-number` is invoked?

Exercise 2: Inserting Strings

Write a new `insert-string` procedure that inserts a string into a list of strings that are in alphabetical order:
> (insert-string "dog" (list "ape" "bear" "cat" "emu" "frog"))
("ape" "bear" "cat" "dog" "emu" "frog")

In case you’ve forgotten, string<=? and string-ci<=? are useful predicates for comparing strings for order.

You may not use the generalized insert procedure in writing this procedure.

**Exercise 3: Generalizing Insertion**

a. Show how to call the generalized insert procedure using lists of strings.

b. Show how to call the generalized insert procedure using lists of numbers.

c. Redefine insert-string so that it uses insert as a helper procedure.

**Exercise 4: Displaying Steps in Insertion Sort**

a. Add calls to the display and newline procedures to the body of the helper in insertion-sort-numbers so that it displays the values of unsorted and sorted, appropriately labeled, at each step of the sorting process.

b. Use the revised insertion-sort-numbers procedure to sort the values 7, 6, 12, 4, 10, 8, 5, and 1.

**Exercise 5: Checking Potential Problems**

Write a test suite (using unit-test.scm) to test the insertion-sort-numbers procedure on some potentially troublesome arguments:

(load "/home/rebelsky/Web/Courses/CS151/2007S/Examples/unit-test.ss")
(begin-tests!)
...
(end-tests!)

a. An empty list

b. A list containing only one element

c. A list containing all equal values

d. A list in which the elements are originally in *descending* numerical order

**Exercise 6: Generalizing Insertion Sort**

Document, write, and test a procedure, (insertion-sort list may-precede?). that generalizes the insertion-sort-numbers procedure.
Exercise 7: Observing insert!

a. Make a copy of the insert! procedure from the reading.

b. Check that it works by using insert! to put the 2 in the correct place in the following vector

(define numbers (vector 1 5 6 7 2 8 0 3))

(Note that solving this step requires that you understand the parameters to insert!.)

c. Extend insert! so that it displays the vector and the position at every step. (Add calls to display and newline in the kernel, before the cond.)

d. Create the numbers vector from step b, and observe what happens when we insert the 2, then the 8, then the 0, then the 3.

e. Make a copy of the insertion-sort! procedure from the reading.

f. Observe the insertion steps in a list of about eight randomly-generated numbers.

(define nums (vector (random 10) (random 10) (random 10) (random 10) (random 10) (random 10) (random 10) (random 10)))