CSE1729 - Introduction to Programming — October 27, 2018

Problem Set 6

1. **(BST Check.)** Define a SCHEME procedure, named `(bst? T)`, that returns true (`#t`) if T has the binary search tree property and false (`#f`) otherwise. Note, this may be trickier than it seems. For example, a tree that has a binary search tree for a left child which contains a value greater than the value at the root is not a binary search tree (even though the left sub-tree is a BST).

2. **(Surgery on Binary Search Trees.)** Suppose that T is a binary search tree as shown on the left hand side of Figure 1. As T is a binary search tree, we must have $b < a$; furthermore, all elements in T_1 are smaller than b, all elements in T_2 lie in the range (a, b), and all elements of T_3 are larger than a. Consider now the operation of *right rotation* of T which re-arranges the root and subtrees as shown. It is easy to check that while this "right rotation" changes the structure of the tree, the resulting tree still satisfies the binary search tree property. (The same can be said of "left rotation.")

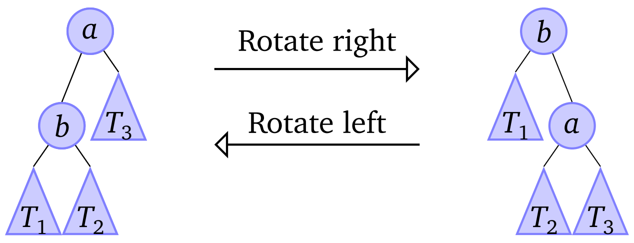

Figure 1: *Right rotation* and *left rotation* of a binary search tree.

The *depth* of an element in a binary tree is the length of the path from the root to the element. It is desirable, for binary search trees, to arrange them so that elements have small depth. (Recall that the *depth* of the tree is the length of the longest path in the tree; equivalently, it is the depth of the "deepest" element.)

Thus, if the depth of T_1 is more than one larger than the depth T_3, a right rotation makes progress toward balancing the tree. Likewise, if the depth of T_3 is more than one more than the depth of T_1, a left rotation seems to be just the thing to improve the tree.

(a) Define a SCHEME procedure, named `(rotate-left T)` which performs a left rotation on its one binary search tree argument, T.

(b) Define a SCHEME procedure, named `(rotate-right T)` which performs a right rotation on its one binary search tree argument, T.

(c) Write a SCHEME procedure called (tree-repair T) which takes, as an argument, a binary search tree. tree-repair should use left and right rotations to try to balance the tree T, as follows:

- Recursively repair each of T's two subtrees. (So, you must build a new tree whose two subtrees are obtained by repairing the two subtrees of T.)
- Once the subtrees have been repaired, examine T. If depth$(T_1) >$ depth$(T_3) + 1$ return the result of rotating T to the right. If, on the other hand, depth$(T_3) >$ depth$(T_1) + 1$ return the result of rotating T to the left.

3. **(Heapsort.)** Write a sorting procedure, named `(hsort elements)` which, given a list of numbers, returns a list of the same numbers in sorted order. Your procedure should do the following:

   - Add all of the elements of the initial list into a heap; be careful to arrange your inserts so that the heap remains balanced. (For this purpose, use the heuristic we discussed in class: always insert into the left child and exchange the order of the children.)
   - Repeatedly *extract-min* (extract the minimum element) from the heap, and return the (sorted) list of elements gathered in this way.

4. **(Fast Access to the Median.)** For some application, we want fast access to the median value of a running list of integers. In order to access the median value of the integers we have been given at any particular time, we will store these integers distributed across two heap data structures. One heap will store roughly half of the values and the other heap will store the remaining integers. We will maintain these heaps in a particular way. One heap will store the lowest integers given in a max-heap (so that the *largest* integer in the heap is at the root). Note, this is different than the implementation of a heap discussed in lecture. The second heap will store the largest integers given so far in a min-heap (i.e. as discussed in lecture).
   Each heap will be stored as a pair consisting of the number of integers in the heap as the car of the pair and the heap in the cdr of the pair. Your integers will be stored in a pair of these heaps. For example, after we are given the integers 6, 3, 1, 11, and 9, we should have the pair of heaps: `((3 6 (3 (1 () ()) ()) ()) 2 9 (11 () ()) ())`
   Recall, that when creating a pair of lists, the SCHEME interpreter will not be able to infer whether the intent was to create a pair of lists or a list of lists. The SCHEME interpreter will always choose to display these as a list of lists. So, the pair of heaps above consists of one heap of three numbers, namely `(6 (3 (1 () ()) ()) ())` and one heap of two numbers, namely `(9 (11 () ()) ())`

   (a) Define a SCHEME procedure, named `(equalize-heaps heap-pair)` which takes a heap pair (in the format outlined above) and checks to see if they differ in size by more than one integer. If so, then the procedure returns a pair of heaps obtained by removing one element from the larger heap and placing it in the smaller heap until the two heaps are equal or only differ in size by one integer.

   (b) Define a SCHEME procedure, named `(add-number x heap-pair)` which adds a new number to our heap pair by inserting the new number, `x`, into the heap of "smaller" values and then using `equalize-heaps` to move element(s) between the heaps in the pair until they have roughly the same number of elements.

(c) Once we have the integers distributed between these two heaps, we can easily access (or compute, if necessary) the *effective median* of the integers we have stored. If the two heaps have an unequal number of elements stored in them, then the root value of the larger heap is the median value. In the case where both heaps have an equal number of elements, then we must compute the *effective median* by computing the average of the roots of the two heaps.

In the example above, one heap has three elements, the other has only two. So, the median value is at the root of the heap with three elements, namely 6.

Note, now we have instant access to the median of this collection of numbers. Moreover, we may add numbers to our heap pair at any time and still have instant access to the updated effective median.

Define a SCHEME procedure, named `(get-median heap-pair)` which returns the effective median of the numbers stored in its single argument, `heap-pair`.