CSE1729 - Introduction to Programming	November 5, 2020
Problem Set 8	

1. This problem concerns ways to represent arithmetic expressions using trees. For this purpose, we will consider 4 arithmetic operations: $+$ and $*$, both of which take two arguments, and $-$ and $^1/_\square$, both of which take one argument. (As an example of how these one-argument operators work, the result of applying the operator $-$ to the number 5 is the number -5; likewise, the result of applying the $^1/_\square$ operator to the number 5 is the number $1/5$.)

   An *arithmetic parse tree* is a special tree in which every node has zero, one, or two children and:

   - each leaf contains a numeric value, and
   - every internal node with exactly two children contains one of the two arithmetic operators $+$ or $*$,
   - every internal node with exactly one child contains one of the two arithmetic operators $-$ or $^1/_\square$.

   (You may assume, for this problem and the next, that when a node has a single child, this child appears as the left subtree.)

   If T is an arithmetic parse tree, we associate a value with T (which we call value(T)) by the following recursive rule:

   - if T has a single (leaf) node, value(T) is equal to the numeric value of the node,
   - if T has two subtrees L and R, and its root is the operator $+$, then value(T) = value(L) + value(R),
   - if T has two subtrees L and R, and its root is the operator $*$, then value(T) = value(L) $*$ value(R),
   - if T has one subtree, S, and its root is the operator $-$, then value(T) $= -$value(S),
   - if T has one subtree, S, and its root is the operator $^1/_\square$, then value(T) $= 1/$ value(S).

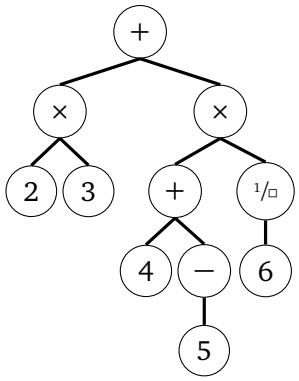
   You can see how to associate with any arithmetic expression a natural arithmetic parse tree. Note that since $-$ and $^1/_\square$ are *unary* operators (take only one argument), you have to give a little thought to how to represent expressions such as $3-5$ or $3/5$. For example, the arithmetic parse tree for the expression

   $$2 \times 3 + \frac{4-5}{6}$$

   is shown in Example 1.

   Write a SCHEME function (`arithvalue T`) which, given an arithmetic parse tree T, computes the value associated with the tree. You may assume that the operators $+$, $\times$, $-$, and $^1/_\square$ appear in the tree as the *characters* `#\+`, `#\*`, `#\-`, and `#\/`. (See the comments at the end of the problem set concerning the SCHEME character type.) To test your code, try it out on the parse tree

```
(define example (list #\+ (list #\*
                                (list 4 '() '())
                                (list 5 '() '()))
```



Example 1: An arithmetic parse tree for the expression $2 \times 3 + (4 + (-5)) \times (1/6)$.

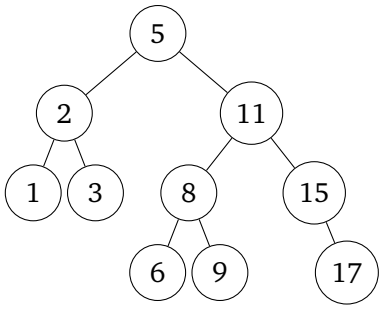
```
(list #\+
      (list #\/ (list 6 '() '()) '())
      (list 7 '() '()))))
```

2. Define a SCHEME procedure, named `(delete-value v T)`, that takes a binary search tree, `T`, and a value, `v`, as arguments and returns the binary search tree that results from removing the node containing the value v. (The tree T should be the first argument.) You may assume that the tree contains no more than one node with value v, but your procedure should be well-behaved if called with a tree T that does not contain the value v: in this case, it should return the tree unchanged. Note that the tree your function returns *must maintain the binary search tree property*. Removing interior (that is, non-leaf) nodes requires a little care. (See Example 2 for an example.) Details are given below.

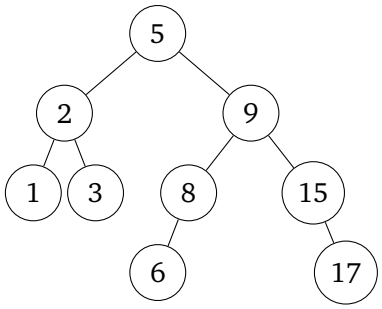
   Deleting nodes from trees The `delete-value` function must remove the one occurrence of v from the input binary search tree T. Naturally, if T has n nodes, the output tree T' has $n-1$ nodes; note that T' must satisfy the binary search tree property (*for every node w, the nodes in the left sub-tree of w hold values smaller than the value held in w and the nodes in the right-subtree of w hold values larger than the value held in w*). There are number of cases that need to be handled separately: the node n may have no sub-trees, exactly one subtree, or two subtrees. (See Example 4.) The first two situations can be handled easily; just remove the node and "replace" it with its single subtree. The third case, illustrated below in Example 3, requires that you restructure the tree a bit to reattach the two "orphaned" subtrees of n in an appropriate way. There are two natural choices: either replace the value at the deleted node with the largest value in the left subtree or the smallest value in the right subtree while removing the corresponding leaf node. In your implementation, please adopt the first of these conventions: specifically, in order to remove a node n with two subtrees, replace the value of n with the *largest* value in the left subtree, and then remove this value from the left subtree. (See the example below.)

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:

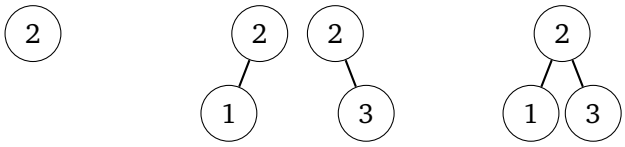
   - Add all of the elements of the initial list into a heap; be careful to arrange your inserts so that



Example 2: An example of a binary search tree.



Example 3: The same binary search tree after the removal of the value 11. Note, we chose to promote the largest value in the left subtree (9) to the node vacated by the value 11.



Example 4: Three possibilities when finding a node to remove. (The middle two cases are treated as one.)

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 if it is less than the value at the root of the heap of "smaller" values and into the heap of "larger" values otherwise. Then using `equalize-heaps` 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`.