CSE1729	Introduction to Programming Final Exam Solutions	May 2, 2017

Short Answers

1. **(10 points.)** Give short Scheme functions or short answers for the following.

   (a) **(3 points.)** Write a Scheme function (convert2vec L) which "converts" a list to a vector. Specifically, given a list L the function should return a vector of the same length as L which contains the elements of L (in the same order). You are free to use the built-in Scheme function **length** which returns the length of a list. Otherwise, however, your solution must work from first principles—you may not use built-in conversion functions.

```
(define (convert2vec L)
  (let ((result (make-vector (length L))))
    (do ((i 0 (+ i 1))
         (current L (cdr current)))
      ((null? current) result)
      (vector-set! result i (car current)))))
```

   (b) **(2 points.)** Give a Scheme function (cube-root x acc) which, given a positive number x, extracts an approximation to $\sqrt[3]{x}$, the cube root of x. Specifically, the function should return a value z so that $|z - \sqrt[3]{x}| <$ acc.

   Your function should use the "bisection method," described as follows: The method maintains two positive numbers, $a < b$, with the property that $a^3 < x < b^3$. Initially the numbers can be set to $a = 0$ and $b = \max(1, x)$. Note that if $|b - a| <$ acc then a can be returned as the final value, as it must be withing acc of the cube root. Otherwise, let $c = (b + a)/2$ and compare c^3 with x. If $c^3 > x$, restart the process with the numbers a and c; otherwise restart the process with the numbers c and b.

```
(define (cube-root x acc)
  (let bisect ((a 0)
               (b (max x 1)))
    (let ((c (/ (+ a b) 2)))
      (cond ((< (- b a) acc) c)
            ((> (* c c c) x) (bisect a c))
            (else (bisect c b))))))
```

   (c) **(2 points.)** Give a Scheme function (**intersection** A B) which, given two lists A and B of numbers, returns a list containing the numbers that appear in *both* A and B. For example

```
> (intersection '(1 3 5 7) '(3 5 4 7))
'(3 5 7)
```

```
(define (intersection A B)
  (define (element x S)
    (and (not (null? S))
         (or (= x (car S))
             (element x (cdr S)))))
  (cond ((null? A) '())
        ((element (car A) B) (cons (car A)
                                   (intersection (cdr A) B)))
        (else (intersection (cdr A) B))))
```

(d) **(3 points.)** Explain, in English, how "binary search" can be used to test for membership in a sorted vector of integers. This is not possible in a sorted SCHEME list of integers; why?

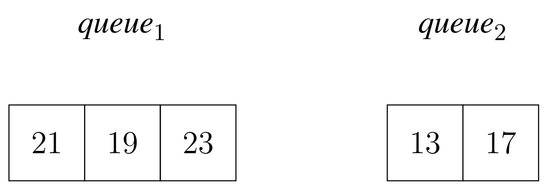
See your notes. SCHEME lists are not "random access": there is no way access an element in the middle without traversing the list.

APPENDABLE QUEUES

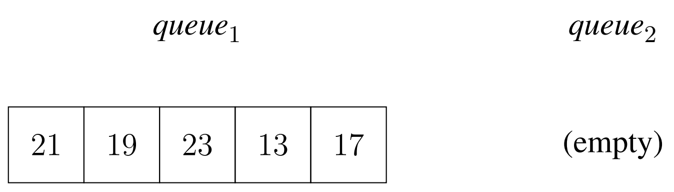
2. **(10 points.)** The classic queue abstract data type supports the operations **enqueue**, **dequeue**, and **empty?**. In this problem we will discuss an *appendable queue* which is comprised of two queues which can be independently operated upon by standard enqueue and dequeue operations. The *appendable queue* has a further operation, **append**, which places the elements of the second queue on the end of the first queue (and removes all elements from the second queue).

To be precise, the appendable queue maintains two queues $queue_1$ and $queue_2$ and offers 7 operations: **enqueue**$_1$, **enqueue**$_2$, **dequeue**$_1$, **dequeue**$_2$, **empty?**$_1$, **empty?**$_2$, and **append**. The operations **enqueue**$_1$, **dequeue**$_1$, and **empty?**$_1$ behave as do the regular queue operations on the first queue $queue_1$. Likewise, the operations with the subscript 2 (e.g., **enqueue**$_2$) operate on the second queue $queue_2$. Finally, the effect of the **append** operation is to move all elements from $queue_2$ onto the end of $queue_1$ (keeping the relative order of the elements in $queue_2$). One possible way to do this using the standard queue operations would be to repeatedly dequeue the first element of $queue_2$ and enqueue the element onto $queue_1$ (until $queue_2$ is emptied).

For example, after the operations **enqueue**$_1$(21), **enqueue**$_2$(13), **enqueue**$_2$(17), **enqueue**$_1$(19), **enqueue**$_1$(23), the queues would contain the elements as pictured below:



Here the left side of the queue is the "front." If the **append** operation were now called, the second queue would be emptied, and the first queue would contain the elements pictured below:



(a) **(2 points.)** Consider an implementation of the appendable queue which uses a standard SCHEME list to maintain each queue and just uses the built-in SCHEME **append** function to carry out **append**. This has the advantage that it is simple to code in SCHEME.

In this implementation, how does the number of elements in the two queues determine the time taken to carry out an **append**? Justify your answer.

It depends on the number of elements in the first queue, as **append** will build a new list starting with the second queue and repeatedly add elements from the end of the first queue onto the front using **cons**.

(b) **(2 points.)** An alternate queue implementation maintains a queue with k elements by placing them in the *first* k slots of a vector. If the two queues are independently maintained in this way, how does the number of elements in the two queues determine the time taken to carry out an **append**? Justify your answer. (You may assume that the vectors are large enough to hold the queues, so no resizing is ever necessary.)

It depends on the number of elements in the second queue, as these elements must be copied from the second vector into the first vector.

(c) **(6 points.)** Give a full implementation of an appendable queue so that all seven operations are efficient, in the sense that the time they take does not depend on the number of elements in the queue(s). (Hint: This is not possible using any of the implementations we have discussed using vectors. Maintain the elements in lists and use destructive set operations to maintain a variable that "points to" the last element of the list.)

Your object should have the structure below:

```
(define (aqueue)
  (let (...)
    (define (empty?1) ...)
    (define (empty?2) ...)
    (define (enqueue1 x) ...)
    (define (enqueue2 x) ...)
    ...
    (define (append) ...)
    ...
    ))
```

Since the code for independently operating on the two queues (e.g., enqueue1 and enqueue2) will be almost identical, it's fine if you just implement one of these and indicate in English how they differ.

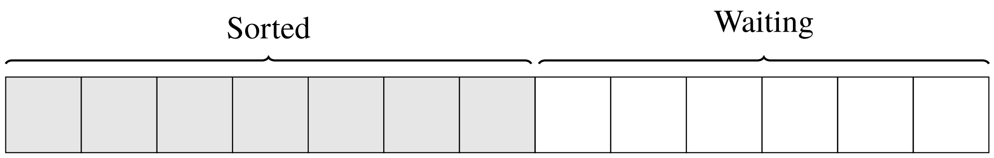
```
(define (aqueue)
  (let ((heads (make-vector 2 '()))
        (tails (make-vector 2 '())))
    (define (empty? q) (null? (vector-ref heads q)))
    (define (enqueue q x)
      (let ((newnode (list x)))
        (if (empty? q)
            (begin (vector-set! heads q newnode)
                   (vector-set! tails q newnode))
            (begin (set-cdr! newnode (vector-ref heads q))
                   (vector-set! heads q newnode)))))
    (define (dequeue q)
      (let ((return (car (vector-ref heads q))))
        (begin
          (vector-set! heads q (cdr (vector-ref heads q)))
          (if (null? (vector-ref heads q))
              (vector-set! tails q '()))
          return)))
    (define (append)
      (if (empty? 0)
          (begin (vector-set! heads 0 (vector-ref heads 1))
                 (vector-set! tails 0 (vector-ref tails 1))
                 (vector-set! heads 1 '())
                 (vector-set! tails 1 '()))
          (begin (set-cdr! (vector-ref tails 0)
                           (vector-ref heads 1))
                 (vector-set! tails 0 (vector-ref tails 1))
                 (vector-set! heads 1 '())
                 (vector-set! tails 1 '()))))
    (define (dispatcher method)
      (cond ((eq? method 'empty?) empty?)
            ((eq? method 'enqueue) enqueue)
            ((eq? method 'dequeue) dequeue)
            ((eq? method 'append) append)))
    dispatcher))
```

Insertion Sort

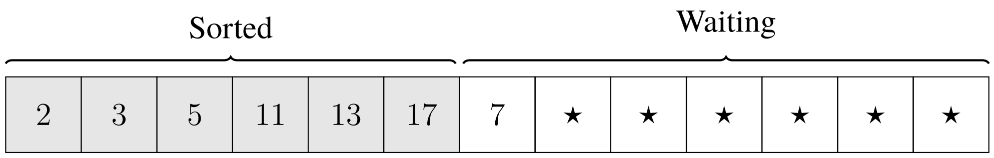
3. **(10 points.)** *Insertion sort* is a famous sorting algorithm which is well-suited for sorting

vectors that are nearly sorted; it is an "in-place" algorithm which is fairly straightforward to code.

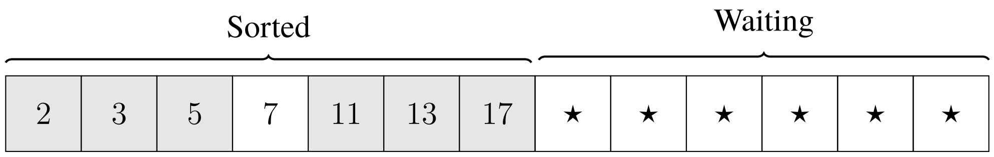
Insertion sort works by conceptually dividing the vector into two parts: a *sorted* part and a *waiting* part. Initially, the sorted part is empty and the waiting part consists of the entire vector. In general, the sorted part will appear as a prefix of the vector and will consist of elements in sorted order. The waiting part will consist of the rest of the vector; elements in the waiting part are not guaranteed to appear in any particular order.



The basic step of the algorithm moves one element from the waiting part to the sorted part, by inserting the element at the correct location in the sorted part. For example, with the waiting and sorted parts as shown below, the algorithm would insert the first waiting element (7 in this case) at the correct position in the sorted part—this will typically involve moving over some elements in the sorted part to make room.



(One step of the algorithm inserts the 7 at the correct location in the sorted part.)



Note that this reduces the length of the waiting part by one and increases the length of the sorted part by one. Once all elements have been inserted into the sorted part, the waiting part is empty and the resulting vector is in sorted order.

(a) **(5 points.)** Write a SCHEME function `isort` which implements insertion sort in a vector. Specifically, given a vector of numbers, the function should return the (same) vector in sorted order. (You might start by writing a SCHEME function `insert` which carries out one insertion step of the algorithm, as described above, starting at a particular position p.)

```
(define (visort elements)
  (define (swap i)
    (let ((temp (vector-ref elements i)))
      (vector-set! elements i (vector-ref elements (+ i 1)))
      (vector-set! elements (+ i 1) temp)))
  (define (shuffle-down p)
    (and (> p 0)
         (< (vector-ref elements p)
            (vector-ref elements (- p 1)))
         (begin (swap (- p 1))
                (shuffle-down (- p 1)))))
  (do ((p 1 (+ p 1)))
    ((= p (vector-length elements)) elements)
    (shuffle-down p)))
```

(b) **(5 points.)** Write an iterative, functional version of insertion sort which operates with SCHEME lists instead of vectors. Your function `(lisort elements)` should take one list `elements` as a parameter, and return a list containing the same elements in sorted order.

To implement insertion sort, I recommend that you write a helper function of the form `(lisort-iterator sorted waiting)` which takes two lists as arguments, the first playing the role of the sorted list, the second playing the role of the waiting list. This function should return the entire sorted list. Once this is written you can simply call `(isort-iterator '() initial-list)` to carry out insertion sort.

```
(define (lisort elements)
  (define (insert x sorted)
    (cond ((null? sorted) (list x))
          ((< x (car sorted)) (cons x sorted))
          (else (cons (car sorted)
                      (insert x (cdr sorted))))))
  (define (lisort-iterator sorted waiting)
    (if (null? waiting)
        sorted
        (lisort-iterator (insert (car waiting) sorted)
                         (cdr waiting))))
  (lisort-iterator '() elements))
```

HEAPS

4. **(10 points.)** Recall that a *heap* is a binary tree which satisfies the *heap condition*: each node of the tree is associated with a numeric value, and the numeric value of any node is less than

those of its children. (You may assume there are no duplicates.) For this problem, use the tree conventions we adopted in class: a node of a tree is represented as a list containing the numeric value of the node and the two subtrees. Please use the convenience functions:

```scheme
(define (make-tree v L R) (list v L R))
(define (value T) (car T))
(define (left T) (cadr T))
(define (right T) (caddr T))
```

(a) **(2 points.)** Write a Scheme function `(insert H x)` which, given a heap `H` and a number `x`, returns the heap that results from inserting the value `x` into `H`. Use the *alternating subtree heuristic* for insert to keep the heap balanced: specifically, make sure that subsequent inserts into the same heap will alternate which subtree is used for recursive insertion.

```scheme
(define (insert x H)
  (cond ((null? H) (make-tree x '() '()))
        ((<= x (value H)) (make-tree x
                                     (right H)
                                     (insert (value H)
                                             (left H))))
       (else (make-tree (value H)
                        (right H)
                        (insert x (left H))))))
```

(b) **(2 points.)** Write a Scheme function `(removemin H)` which, given a heap `H`, returns the heap that results from removing the minimum value of *H*.

```scheme
(define (remove-min H)
  (cond ((null? (left H)) (right H))
        ((null? (right H)) (left H))
        ((< (value (left H)) (value (right H)))
         (make-tree (value (left H))
                    (remove-min (left H))
                    (right H)))
       (else (make-tree (value (right H))
                        (left H)
                        (remove-min (right H))))))
```

(c) **(2 points.)** Write a Scheme function `(sorted-stream H)` which, given a heap *H*, returns the stream associated with the elements of *H* appearing in sorted order. (Hint: The first element of the stream is the minimum element in the heap; what determines the rest of the stream?)

```
   (define (extraction-stream H)
      (if (null? H) '())
      (cons (value H)
            (delay (extraction-stream (removemin H)))))
```

(d) **(4 points.)** Write a SCHEME object creator function `(heapify L)` which takes one parameter—a list *L* of numbers—and returns a heap object with the methods `empty?`, **`min`**, and `removemin` that operate on a heap constructed from the numbers in *L*. The idea is that your object creates a heap when it is initially called using the elements of *L*; once the heap is created, it can be (destructively) acted upon by the methods. Note that your object does not need to provide an insert method, but you will need to use this anyway so that you can initially populate the heap with the elements of *L*. Of course, you do not need to re-write code from your previous problems.

Specifically, your object should have the form

```
(define (heapify L)
  ...
  (let ((...))                ;; internal variables
    (define (empty?) ...)     ;; heap methods
    (define (min x) ...)
    (define (removemin) ...)
    (define (dispatcher ...) ...) ;;the dispatcher
    dispatcher))
```

```
(define (heapify L)
  (define (make-tree v L R) (list v L R))
  (define (value T) (car T))
  (define (left T) (cadr T))
  (define (right T) (caddr T))
  (define (insert x H)
    (cond ((null? H) (make-tree x '() '()))
          ((<= x (value H)) (make-tree x
                                        (right H)
                                        (insert (value H)
                                                (left H))))
          (else (make-tree (value H)
                           (right H)
                           (insert x (left H))))))
  (define (populate remnant)
    (if (null? remnant)
        '()
        (insert (car remnant)
                  (populate (cdr remnant)))))
  (define (remove-min-general H)
    (cond ((null? (left H)) (right H))
          ((null? (right H)) (left H))
          ((< (value (left H)) (value (right H)))
           (make-tree (value (left H))
                      (remove-min-general (left H))
                      (right H)))
          (else (make-tree (value (right H))
                           (left H)
                           (remove-min-general (right H))))))
  (let ((heap (populate L)))
    (define (min) (value heap))
    (define (removemin) (set! heap (remove-min-general heap)))
    (define (empty) (null? heap))
    (define (dispatcher method)
      (cond ((eq? method 'empty) empty)
            ((eq? method 'min) min)
            ((eq? method 'removemin) removemin)))
    dispatcher))
```

The Set ADT

5. **(10 points.)** Recall the Set abstract data type, which must implement **insert**, **member?**, and **empty?**. Recall that a *binary-search tree* is a binary tree which satisfies an extra condition: all elements appearing in the left subtree of a given node have values less than the value of the node; likewise, all elements appearing in the right subtree of a node have values that are larger than the value of the node. (You may assume there are no duplicates.) For this problem, use the tree conventions we adopted in class: a node of a tree is represented as a list containing the numeric value of the node and the two subtrees. Please use the convenience functions:

```
(define (make-tree v L R) (list v L R))
(define (value T) (car T))
(define (left T) (cadr T))
(define (right T) (caddr T))
```

   (a) **(3 points.)** Write a Scheme function `(insert x T)` which, given a binary search tree T and an element x, returns the binary search tree that results from inserting x into T.

```
(define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((< x (value T)) (make-tree (value T)
                                   (insert x (left T))
                                   (right T)))
        (else (make-tree (value T)
                         (left T)
                         (insert x (right T))))))
```

(b) **(3 points.)** Write an entire Scheme object (`set-as-bst`) which implements the Set ADT using binary search trees. (Of course, you do not need to re-implement `insert`—just indicate how you use the code.) Your object should have the form

```
(define (set-as-bst)
  (let ((...))                   ;; internal variables
    (define (empty?) ...)        ;; ADT methods
    (define (member? x) ...)
    (define (insert x) ...)
    (define (dispatcher ...) ...) ;;the dispatcher
    dispatcher))
```

```
(define (set-as-bst)
  (let ((bst '()))
    (define (bst-insert x T)
      (cond ((null? T) (make-tree x '() '()))
            ((< x (value T)) (make-tree (value T)
                                        (bst-insert x (left T))
                                        (right T)))
            (else (make-tree (value T)
                             (left T)
                             (bst-insert x (right T))))))
    (define (bst-member x T)
      (cond ((null? T) #f)
            ((= x (value T)) #t)
            ((< x (value T)) (bst-member x (left T)))
            (else (bst-member x (right T)))))
    (define (insert x)
      (set! bst (bst-insert x bst)))
    (define (member? x) (bst-member x bst))
    (define (empty?) (null? bst))
    (define (dispatcher method)
      (cond ((eq? method 'insert) insert)
            ((eq? method 'member?) member?)
            ((eq? method 'empty?) empty?)))
    dispatcher))
```

(c) **(4 points.)** Consider adding to the SET ADT a further operation **shift**, so that **shift**(k) adds the value k to all elements currently held in the set. Explain how to update your object to provide this functionality. For full credit, the time taken to carry out `(shift k)` should not depend on the number of elements in the set. (Hint: Perhaps there is a way that does not involve changing the actual values stored in the tree? Be careful! You may have insertions after a shift.)

```
(define (set-as-bst)
  (let ((bst '())
        (offset 0))
    (define (bst-insert x T)
      (cond ((null? T) (make-tree x '() '()))
            ((< x (value T)) (make-tree (value T)
                                        (bst-insert x (left T))
                                        (right T)))
            (else (make-tree (value T)
                             (left T)
                             (bst-insert x (right T))))))
    (define (bst-member x T)
      (cond ((null? T) #f)
            ((= x (value T)) #t)
            ((< x (value T)) (bst-member x (left T)))
            (else (bst-member x (right T)))))
    (define (insert x)
      (set! bst (bst-insert (- x offset) bst)))
    (define (member? x) (bst-member (- x offset) bst))
    (define (empty?) (null? bst))
    (define (shift k) (set! offset (+ offset k)))
    (define (dispatcher method)
      (cond ((eq? method 'insert) insert)
            ((eq? method 'shift) shift)
            ((eq? method 'member?) member?)
            ((eq? method 'empty?) empty?)))
    dispatcher))
```