CSE1729 - Introduction to Programming March 30, 2016

Prelim 2 Solutions

1. (**10 points.**) Define SCHEME functions with the following specifications.

   (a) (**3 points.**) The built-in SCHEME function `max` returns the maximum of two numbers. Define a SCHEME function `listmax` which takes a list of numbers as input, and returns their maximum.

```
(define (listmax numbers)
  (if (null? (cdr numbers))
      (car numbers)
      (max (car numbers)
           (listmax (cdr numbers)))))
```

   (b) (**3 points.**) Define a SCHEME function `filter` which takes a list *L* and a number *k* as inputs and returns a list containing the numbers in *L* that are less than *k*. For example `(filter '(1 5 2 6 3 7) 4)` should return `(1 2 3)`, as these are the elements of `(1 5 2 6 3 7)` that are less than 4.

```
(define (filter L k)
  (cond ((null? L) '())
        ((< (car L) k) (cons (car L)
                             (filter (cdr L) k)))
        (else (filter (cdr L) k))))
```

   (c) (**4 points.**) Define a SCHEME function `treesize` which, given a binary tree, returns the *number* of nodes in the tree.

```
(define (treesize T)
  (if (null? T)
      0
      (+ 1
         (treesize (left T))
         (treesize (right T)))))
```

2. (**10 points.**) Recall that a *heap* is a binary tree with the property that for every node of the tree, its value is (strictly) smaller the values of its children. Thus, the smallest value of a heap always appears at the root.

   (a) (**5 points.**) Given a heap `H` and a number `v`, define a SCHEME function that properly inserts `v` into `H`; your function should return the resulting heap. Use the alternating subtree heuristic we discussed in class, so that elements are inserted into subtrees by insertion into the left subtree after *exchanging* the two subtrees.

```
(define (heapinsert v H)
  (cond ((null? H) (maketree v '() '()))
        ((< v (value H)) (maketree v
                                   (heapinsert (value H)
                                               (right H))
                                   (left H)))
        (else (maketree (value H)
                        (heapinsert v (right H))
                        (left H)))))
```

(b) (**1 points.**) What's the point of the "alternating subtree heuristic" discussed above? (That is, why not just insert the element into the left subtree?)

This helps to keep the heap balanced, so that it is *shallow*. By alternating between the two subtrees, we can ensure that we have inserted the same number of elements into the two subtrees (upto ± 1).

(c) (**1 points.**) How does the "alternating subtree heuristic" compare with the policy of inserting the element into the *smaller* of the two subtrees?

This would give the identical result (depending on how you break ties) unless values are removed from the heap, in which case the "smaller" heuristic may do a better job of balancing. It would require maintaining the size of each subtree, however, or computing this.

(d) (**3 points.**) Define a SCHEME function `heapify` which, given a list of numbers (call it `numbers`), constructs and returns a heap from the numbers in the list.

```
(define (heapify numbers)
  (define (heapify-acc remaining partialheap)
    (if (null? remaining)
        partialheap
        (heapinsert (car remaining)
                    (heapify-acc (cdr remaining)
                                 partialheap))))
  (heapify-acc numbers '()))
```

or just

```
(define (heapify L)
  (if (null? L)
      '()
      (insert (car L)
              (heapify (cdr L)))))
```

3. (**10 points.**) Consider the problem of finding the *kth smallest element* of a given list of numbers—that is, the number that would appear in the kth position if the list was placed in sorted order. Of course, one obvious way to solve the problem is to sort the list and then return the kth element.

However, there is a nice recursive algorithm for this process which we will call **QuickSelect**; its operation is very like **QuickSort** and, in general, it finds the kth element of an (unsorted) list more quickly than completely sorting the list.

To recap: Given a number k and a list L, **QuickSelect** should return the kth smallest element of L (that is, the kth element of the list that would result if L were sorted). A detailed description of the **QuickSelect** algorithm:

QuickSelect(k, L):

- Select a *pivot* element p from the list L. (p may be any element of the list.)
- **Partition** the list L around the element p. This results in two lists, *Small* and *Big*: the list *Small* contains all elements smaller than p; the list *Big* contains all elements larger than p.
- Let ℓ be the number of elements in *Small*. (Note then that p is the $\ell + 1$st element of the original list when placed in sorted order because there are exactly ℓ elements less than p.)
- If $\ell + 1 = k$, return p.
- If $k \leq \ell$, return **QuickSelect**$(Small, k)$.
- If $\ell + 1 < k$, return **QuickSelect**$(Big, k - \ell - 1)$.

(a) (**4 points.**) Give a full implementation of `partition`, a SCHEME function which takes a list `numbers` and a pivot value `pivot`. It should return a pair, consisting of two lists `smaller` and `larger` which contain the elements that are smaller than and larger than the pivot, respectively. (N.b., this is exactly the same function used in **QuickSort**!)

```
(define (partition L p)
  (define (partition-acc remaining partial-small partial-large)
    (cond ((null? remaining) (cons partial-small partial-large))
          ((< (car remaining) p) (partition-acc
                                  (cdr remaining)
                                  (cons (car remaining)
                                        partial-small)
                                  partial-large))
          (else (partition-acc (cdr remaining)
                               partial-small
                               (cons (car remaining)
                                     partial-large)))))
  (partition-acc L '() '()))
```

(b) (**4 points.**) Give a full implementation of **QuickSelect**. You may use the `partition` function from the previous problem, even if you did not solve it. (Hint: I recommend that you start with a `let*` statement to set the pivot to be the first element of the list, call `partition`, and determine the length of the part of the partition containing "small" elements—you may use the built-in `length` function for this purpose.)

```
(define (quickselect k L)
  (let* ((pivot (car L))
         (parts (partition (cdr L) pivot))
         (small (car parts))
         (large (cdr parts))
         (numsmall (length small)))
    (cond ((= k (+ numsmall 1)) pivot)
          ((<= k numsmall) (quickselect k small))
          (else (quickselect (- k (+ numsmall 1)) large)))))
```

(c) (**2 points.**). Note that both **QuickSort** and **QuickSelect** use a pivot to partition the elements of the list. However, there is an important difference in how they recurse on the pieces. Explain, and indicate why this might make **QuickSelect** faster than **QuickSort**.

Note that **QuickSort** makes a recursive call on *both* parts of the partition, whereas **QuickSelect** makes a recursive call on only one of them. If you are lucky, and partitions always evenly divide the set, each recursive call to **QuickSelect** reduces the size of the list by a factor two.

4. (**10 points.**) Counting small values in ordered trees.

(a) (**3 points.**) Define a SCHEME function `countlessthan` which, given a binary search tree T and a number k, returns the *number of elements* in T less than or equal to k. Observe that if $k <$ `value`(T) then none of the elements in `(right T)` are less than k—use this idea to optimize your code.

```
(define (countlessthan T k)
  (cond ((null? T) 0)
        ((= (value T) k) (+ 1 (countlessthan (left T) k)))
        ((> (value T) k) (countlessthan (left T) k))
        (else (+ (countlessthan (left T) k)
                    (countlessthan (right T) k)
                    1))))
```

(b) (**3 points.**) Consider a fancy adaptation of the binary search tree concept in which each node additionally stores the *size* of the whole tree rooted at the node (*size* here means the number of nodes). Thus, each node actually stores two numbers: the data value (which is an arbitrary number) and a second number equal to the size of the tree rooted at the node. We can adapt our node data structure to maintain this new information by adding to the list: in particular, each node is now represented as

```
(value left right totalsize)
```

and we may add the "convenience" function

```
(define (size T) (if (null? T) 0 (cadddr T)))
```

(The other convenience functions can retain their usual definitions, except for `maketree` which must be adapted to take four parameters.) Of course, in order to keep the size data up to date, they have to be updated appropriately during insertion. Show how to rewrite the standard binary search tree `insert` function to additionally update the size registers. Specifically, define a SCHEME function `insert` which, given a binary search tree T with correct size data (as described above) and a value v, returns the tree that results by inserting v into T—of course, the returned tree should have correct size data!

```
(define (insert v T)
  (cond ((null? T) (make-tree v '() '() 1))
        ((= v (value T)) T)
        ((< v (value T)) (make-tree (value T)
                                    (insert v (left T))
                                    (right T)
                                    (+ 1 (size T))))
        ((> v (value T)) (make-tree (value T)
                                    (left T)
                                    (insert v (right T))
                                    (+ 1 (size T))))))
```

(c) (**4 points.**) Finally, write a SCHEME function `countlessthan` which, given a fancy binary search tree (in which all the size registers indeed correctly hold the tree sizes) and a number k, returns the number of elements less than k. Show how to use the `size` function to give a fast implementation that avoids exploring certain subtrees.

```
(define (countlessthan T k)
  (cond ((null? T) 0)
        ((<= k (value T)) (countlessthan (left T) k))
        ((>  k (value T)) (+ (size (left T))
                             1
                             (countlessthan (right T) k)))))
```