CSE 1729 - Introduction to Principles of Programming — October 31, 2017

Laboratory Assignment 9

Objectives

- Work with symbol frequencies in lists
- Work with heaps
- Learn a new way to sort

Activities

1. Define a Scheme function `(num-occurs sym lst)`, that takes two parameters: a symbol `sym` and a list `lst`; it evaluates to the number of times the symbol appears in the list. The list might be nested, but you do not count any symbols within nested lists. For example

```
> (num-occurs 'uh-huh '(thats the way uh-huh uh-huh i like it uh-huh uh-huh))
4
> (num-occurs 'a '(a b c (not (c b a))))
1
```

2. Define a Scheme function, named `freq-list`, which takes a list of symbols as a parameter and returns a list of symbol-frequency pairs. That is, the function will produce a list of pairs where the first value in the pair is a symbol and the second value in the pair is the frequency (i.e. the number of times) that symbol appears in the list. There will be only one pair for each distinct symbol in the list.

   There are multiple approaches possible, but you should only count the number of occurences of each symbol once (Hint: you can use `num-occurs`.)

   Example:

```
> (freq-list
    '(thats the way uh-huh uh-huh i like it uh-huh uh-huh
      thats the way uh-huh uh-huh i like it uh-huh uh-huh
      thats the way uh-huh uh-huh i like it uh-huh uh-huh
      thats the way uh-huh uh-huh i like it uh-huh uh-huh))
((it . 4)(like . 4)(i . 4)(uh-huh . 16)(way . 4)(the . 4)(thats . 4))
```

   The order of the elements in what `freq-list` returns is unimportant – it will depend on your implementation.

3. Build the logic for a heap structure that stores symbol-frequency-pairs, making it easy to find the least frequent symbols stored in the heap. Most commonly, heaps are used to store pairs consisting of one items of any type, and one number to be used to compare with other pairs when looking for the minimum value. As an example, consider pairs that contain tasks and

their deadlines; we would like to store them so we access the earliest deadline first – we want to extract the pair with the minimum deadline value.

Specifically, you should write Scheme procedures for heap as in lecture: `create-heap`, `h-min`, `left`, `right`, `insert`,`insert-list`, and `remove-min`. The difference between these and the code we discussed in lecture is that each entry consists of a pair, and the second element of the pair is a number that is used in ordering the heap. It will make your code clearer if you define two functions for the pairs that you put in the heap, `value` and `weight`. Given these, the heap characteristic is that for each heap, the weight of all of the nodes in the left and right subtrees will be greater than or equal to than the weight at the root, i.e. if H is a heap with non-null subtrees,

```
>(> (weight (h-min H))(weight (h-min (left H))))
#f
>(> (weight (h-min H))(weight (h-min (right H))))
#f
```

Note that more than one node in the tree can have the same weight.

The specific functions you need to write and test are:

(a) `(create-heap vw-pair left right)` evaluates to the heap with `vw-pair` a pair consisting of a value (anything) and a weight, which must be a number.

(b) `(h-min heap)` evaluates to the value-weight pair at the root, that is, the pair with the smallest weight in the heap,

(c) `(left heap)` evaluates to the left child of the heap,

(d) `(right heap)` evaluates to the right child of the heap,

(e) `(insert vw-pair heap)` evaluates to the heap resulting from inserting the value-weight pair `vw-pair`,

(f) `(insert-list-of-pairs vw-pair-list heap)` evaluates to the heap resulting from inserting all of the value-weight pairs from the list `vw-pair-list` into the heap.

(g) `(remove-min heap)` evaluates to the heap resulting when the value-weight pair with minimum weight (at the root) is removed.

Remember that the results of `insert` and `remove-min` are heaps, which means that the weight in the value-weight pair at the root is less than or equal to the weights of all of the value-weight pairs in its subheaps. Also, remember to alternate which subtree gets inserted into (see lecture slides) to maintain balance.

4. Heaps can also be used to sort sequences of pairs by weight! Suppose I have a list of value-weight pairs in any order, I can use my heap code to sort them by

    1. Insert each of the list of pairs in the heap,
    2. Successively add the minimum to a list, and remove the minimum from the heap, until the heap is empty.

    You have a function above (`insert-list-of-pairs` that can do the first part. So what you need to do is write a Scheme function `(get-in-order heap)` that uses `h-min` and `remove-min` to get the list of pairs sorted by weight. Once this works, you can implement heapsort as

```
(define (heapsort pair-list)
  (get-in-order (insert-list-of-pairs pair-list '())))
```

5. Combine the results from questions 2 and 4 by doing the following. Given a list of symbols, use freq-list to generate the associated list of symbol-frequency pairs. Use heapsort to sort that list. Be sure and generate interesting examples (lists of symbols with repeats so that not all have the same frequencies, eg.), and show what `(freq-list your-list)` and `(heapsort (freq-list your-list))` evaluate to.

   For example:

```
> (freq-list '(row row your boat boat boat row your boat boat boat))
((boat . 6) (your . 2) (row . 3))
> (heapsort
    (freq-list '(row row your boat boat boat row your boat boat boat)))
((your . 2) (row . 3) (boat . 6))
```