| CSE1729 | Introduction to Programming<br>Problem Set 10 Solutions | April 25, 2017 |
|---|---|---|

**Constructing Huffman trees**

Recall the discussion from class on Huffman trees. In particular, to construct an optimal encoding tree for a family of symbols $\sigma_1, \ldots, \sigma_k$ with frequencies $f_1, \ldots, f_k$, carry out the following algorithm:

1. Place each symbol $\sigma_i$ into its own tree; define the weight of this tree to be $f_i$.
2. If all symbols are in a single tree, return this tree (which yields the optimal code).
3. Otherwise, find the two current trees of minimal weight (breaking ties arbitrarily) and combine them into a new tree by introducing a new root node, and assigning the two light trees to be its subtrees. The weight of this new tree is defined to be the sum of the weights of the subtrees. Return to step 2.

As an example, consider Huffman encoding a long English text document:

- You would begin by computing the frequencies of each symbol in the document. This would produce a table, something like the one shown below.

| Symbol | Frequency |
|---|---|
| a | 2013 |
| b | 507 |
| c | 711 |
| ⋮ | ⋮ |

  Here the "frequency" is the number of times the symbol appeared in the document. (If you prefer, you could divide each of these numbers by the total length of the document; in that case, you could think of the frequencies as probabilities. This won't change the results of the Huffman code algorithm.)

- Following this, you can apply the Huffman code algorithm above: this will produce a "Huffman code tree." The purpose of this tree is to associate a "codeword" with each symbol. Specifically, the path from the root to a given symbol can be turned into a codeword by treating every step to a left child as a zero and every step to a right child as a one. In the figure below, the symbol $\alpha$ would be associated with the codeword 100.

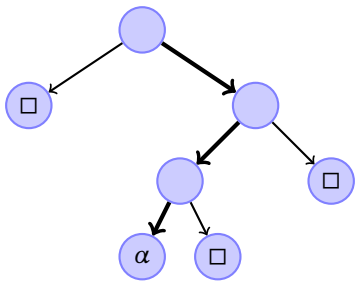


Figure 1: A Huffman tree; the codeword associated with $\alpha$ is 100. The □ placeholders represent other symbols.

- Finally, you can "encode" the document by associating each symbol in the document with the corresponding codeword. Note that this will turn the document into a long string of 0s and 1s.
- Likewise, you can "decode" the encoded version of a document, by reading the encoded version of the document from left to right, and following the path it describes in the Huffman tree. Every time a symbol is reached, the process starts again at the root of the tree.

**Strings and characters in SCHEME** SCHEME has the facility to work with strings and characters (a *string* is just a sequence of characters). In particular, SCHEME treats *characters* as atomic objects that evaluate to themselves. They are denoted: `#\a`, `#\b`, .... Thus, for example,

```
> #\a
#\a
> #\A
#\A
> (eq? #\A #\a)
#f
> (eq? #\a #\a)
#t
```

The "space" character is denoted `#\space`. A "newline" (or carriage return) is denoted `#\newline`.

A *string* in SCHEME is a sequence of characters, but the exact relationship between strings and characters requires an explanation. A string is denoted, for example, `"Hello!"`. You can build a string from characters by using the `string` command as shown below. An alternate method is to use the `list->string` command, which constructs a string from a list of characters, also modeled below. Likewise, you can "explode" a string into a list of characters by the command `string->list`:

```
> (string #\S #\c #\h #\e #\m #\e)
"Scheme"
> (list->string '(#\S #\c #\h #\e #\m #\e))
"Scheme"
> (string->list "Scheme")
(#\S #\c #\h #\e #\m #\e)
> "Scheme"
"Scheme"
```

Note that strings, like characters, numbers, and Boolean values, evaluate to themselves. Finally, the SCHEME function `string-append` concatenates (or glues together) two strings. For example

```
> (string-append "book" "keeper")
"bookkeeper"
> (string-append "Comic" "␣" "relief")
"Comic␣relief"
```

(Note the appearance of the string `"␣"` containing a space as the second argument to this function.)

---

1. Write a SCHEME function which, given a list of characters and frequencies, constructs a Huffman encoding tree. You may assume that the characters and their frequencies are given in a list of pairs: for example, the list

   ```
   ((#\a . 2013) (#\b . 507) (#\c . 711))
   ```

   represents the 3 characters $a$, $b$, and $c$, with frequencies 2013, 507, and 711, respectively. Given such a list, you wish to compute the tree that results from the above algorithm. I suggest that you maintain nodes of the tree as lists: internal nodes can have the form

```
(internal 0-tree 1-tree)
```

where `internal` is a token that indicates that this is an internal node, and `0-tree` and `1-tree` are the two subtrees; leaf nodes can have the form

```
(s () ())
```

where `s` is the character held by the leaf. Note that you will have to use the SCHEME `quote` command to construct internal nodes: for example, to construct an internal node with the two subtrees `0-tree` and `1-tree`, you could use the procedure below

```
(define (make-internal-node 0-tree 1-tree)
  (list 'internal 0-tree 1-tree))
```

Note that when you traverse a Huffman coding tree, you can determine if a given node is an internal node by deciding if the `car` of the list associated with that node is the token `internal`.

2. Define a SCHEME function that takes, as input, a Huffman coding tree and outputs a list containing the elements at the leaves of the tree along with their associated encodings as a string over the characters `#\0` and `#\1`. For example, given the tree of Figure 2, your function should return the list

```
((#\a . "0") (#\b . "10") (#\c . "11")) .
```

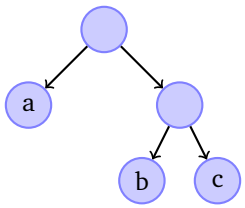


Figure 2: A Huffman tree yielding an encoding of the three symbols a, b, and c.

3. Define a SCHEME function `encode` which takes, as input, a string and a list of characters and frequencies (as in problem 1) and encodes the string into a 0/1-string using Huffman coding. (You may assume that every character in the string is indeed in the list of character/frequency pairs.)

4. Define a SCHEME function `decode` which takes, as input, a string over the symbols 0/1 and a Huffman coding tree and "decodes" the string according to the tree. (It should return a string.)

5. Finally, show how to package all of the functionality you have created in this assignment as a `Huffman-code` object. Here's the idea.

   - Define a function `H-code-object` which takes, as an argument, a list of character/frequency pairs. It should return a dispatcher that yields two methods: `encode`, which maps a character string to a string of zeros and ones, and `decode`, which maps a string of zeros and ones to a character string. You might also implement a "reminder" method which simply returns the set of allowable characters (the ones that appeared in the original character/frequency list).
   - When your object is called (with a collection of symbol/frequency pairs), it should use a `let` (or `let*`) to immediately create the appropriate Huffman tree and the associated encoding table. Then it should return the methods for encoding and decoding as usual. Thus the rough structure of your object should be something like this:

```
(define (H-code-object frequencies)
  (define (make-internal-node 0-tree 1-tree)
    (list 'internal 0-tree 1-tree))
  (define (value T) (cadr T))
  ;; Other tree maintenance functions.

  ;; The function for constructing the Huffman tree.
  (define (construct-tree frequencies) ...)

  ;; The function for constructing the encoding table.
  ;; Note that it takes a huffman tree as a parameter.
  (define (construct-encoding-list huffman) ...)

  ;; Finally, the body of the object, where we create
  ;; the tree and the encoding list
  (let* ((decode-tree (construct-tree frequencies))
         (encode-list (construct-encoding-list decode-tree)))

    ;;The encoder
    (define (encode text) ...)

    ;; The decoder
    (define (decode text) ...)

    ;; The dispatcher
    (define (dispatcher method)
      (cond ((eq? method 'encode) encode)
            ((eq? method 'decode) decode)))
    dispatcher
    ))
```

To maintain the trees during the Huffman coding algorithm, we implement a priority queue as a heap.

```
(define (make-pqueue comparator<?)
  (let ((theHeap '())
        (heapsize 0))
    ;; As we will maintain the priority queue as a heap, we lay out
    ;; some basic tools for manipulating trees.
    ;; Nodes in these tree have the structure
    ;; (value left-tree right-tree).
    (define (make-tree value left right)
      (list value left right))
    (define (value T) (car T))
    (define (left T)  (cadr T))
    (define (right T) (caddr T))
    ;; Heap tools
    (define (combine Ha Hb)
      (cond ((null? Ha) Hb)
            ((null? Hb) Ha)
            ((comparator<? (value Hb) (value Ha))
               (make-tree (value  Hb) Ha
                          (combine  (left Hb) (right Hb))))
            (else (make-tree (value Ha)
                             (combine (left Ha) (right Ha)) Hb))))
    (define (heap-insert x workHeap)
      (cond ((null? workHeap) (make-tree x '() '()))
            ((comparator<? x (value workHeap))
             (make-tree x
                        (heap-insert (value workHeap)
                        (right workHeap))
                        (left workHeap)))
            (else
             (make-tree (value workHeap)
                        (heap-insert x (right workHeap))
                        (left workHeap)))))
    ;; Method definitions
    (define (empty?) (null? theHeap))
    (define (insert x)
      (begin
        (set! theHeap (heap-insert x theHeap))
        (set! heapsize (+ heapsize 1))))
    (define (extract-min)
      (let ((smallest (value theHeap)))
        (begin (set! theHeap (combine (left theHeap)
                                      (right theHeap)))
               (set! heapsize (- heapsize 1))
               smallest)))
    ;; Finally, the method dispatcher
    (lambda (method)
      (cond ((eq? method 'empty) empty?)
            ((eq? method 'insert) insert)
            ((eq? method 'extract-min) extract-min)
            ((eq? method 'size) heapsize)
```

```
          ))))
```

With this bit of infrastructure behind us, we are ready for the Huffman code object.

```
(define (H-code-object frequencies)
  ;; Tree maintenance functions.
  ;; Our Huffman tree shall have two sorts of nodes:
  ;; internal nodes, with the format
  ;; ('internal 0-tree 1-tree), and leaf nodes, with the format
  ;; ('leaf character).
  (define (make-internal-node 0-tree 1-tree)
    (list 'internal 0-tree 1-tree))
  (define (make-leaf-node value)
    (list 'leaf value))
  (define (leaf? T) (eq? (car T) 'leaf))
  (define (value T) (cadr T))
  (define (subtree T c)
    (cond ((eq? c #\0) (cadr T))
          ((eq? c #\1) (caddr T))))
  ;; Now the routine for constructing the Huffman tree.
  (define (construct-tree frequencies)
    ;; The function uses a priority queue to maintain the
    ;; weighted trees. We maintain weighted trees a functions
    ;; f so that f('tree) is the tree, and f('weight) is the weight.
    ;; (Recall that the weight of the tree will be the
    ;; sum of the frequencies of the symbols in the tree.)
    ;; These functions are the objects actually put into the priority
    ;; queue. We begin by setting down some functions to operate on
    ;; these weighted trees:
    (define (make-weighted-tree T weight)
      (lambda (method)
        (cond ((eq? method 'tree) T)
              ((eq? method 'weight) weight))))
    (define (weighted-tree-comparator wTa wTb)
      (<= (wTa 'weight) (wTb 'weight)))
    (define (merge-weighted-trees wTa wTb)
      (make-weighted-tree
       (make-internal-node (wTa 'tree) (wTb 'tree))
       (+ (wTa 'weight) (wTb 'weight))))
    ;; Set up the priority queue:
    (let ((pq (make-pqueue weighted-tree-comparator)))
      ;; a routine to fill up the priority queue.
      ;; It returns the filled pq.
      (define (populate-queue char-freq-list)
        (if (null? char-freq-list)
            pq
            (begin
              ((pq 'insert) (make-weighted-tree
                             (make-leaf-node (caar char-freq-list))
                             (cdar char-freq-list)))
              (populate-queue (cdr char-freq-list)))))
      ;; a routine to 'process' the priority queue. It repeatedly
      ;; pulls off the smallest two elements of the queue, combines
```

```
    ;; them into a single weighted tree, and adds
    ;; this back to the queue. It ends when there is only one
    ;; tree left in the priority queue and returns it.
    (define (process-queue pq)
      (if (= (pq 'size) 1)
          (((pq 'extract-min)) 'tree)
          (begin (let* ((first ((pq 'extract-min)))
                        (second ((pq 'extract-min))))
                   ((pq 'insert) (merge-weighted-trees second
                                                       first)))
                 (process-queue pq))))
    ;; top-level body of construct-tree. It fills the priority queue
    ;; and then empties it to construct the Huffman tree bottom-up.
    (process-queue (populate-queue frequencies))))

;; Create the encoding table
(define (construct-encoding-list huffman)
  (define (extract-encoding subT path-string)
    (if (leaf? subT)
        (list (cons (value subT) (reverse path-string)))
        (append (extract-encoding (subtree subT #\0)
                                  (cons #\0 path-string))
                (extract-encoding (subtree subT #\1)
                                  (cons #\1 path-string)))))
  (extract-encoding huffman '()))
;; The code for using the encoding table to lookup an encoding.
(define (lookup-encoding character e-list)
  (cond ((null? e-list) "")
        ((eq? character (caar e-list)) (cdar e-list))
        (else (lookup-encoding character (cdr e-list)))))

(let* ((decode-tree (construct-tree frequencies))
       (encode-list (construct-encoding-list decode-tree)))

  ;;The encoder
  (define (encode text)
    (list->string (foldr (lambda
                           (c t)
                           (append (lookup-encoding c encode-list)
                                   t))
                         '()
                         (string->list text))))
  ;; The decoder
  (define (decode text)
    (define (decode-iterator text-list subT)
      (cond ((leaf? subT) (cons (value subT)
                                (decode-iterator text-list
                                                 decode-tree)))
            ((null? text-list) '())
            (else (decode-iterator (cdr text-list)
                                   (subtree subT
```

```
                                                (car text-list))))))
  (list->string (decode-iterator (string->list text)
                                 decode-tree)))
;; The dispatcher
(define (dispatcher method)
  (cond ((eq? method 'encode) encode)
        ((eq? method 'decode) decode)))
;; The object-creator
dispatcher
))
```