Fall CS 145 Final Review Guide.

֍ Prof: Gordon Cormack
֍ ISA: Ashish Mahto
֍ Author: Teresa Kang
֍ Edit: David Duan - j32duan@edu.uwaterloo.ca

==========================================================================================

*Topic.* AVL-SET

*Note.*

Given avl-cs145.rkt (implementation of AVL tree), define set ADT.

*Implementation.* avl-cs145.rkt

```
<rkt>
  empty               ;; an empty tree
  (node-left t)       ;; left subtree of t
  (node-right t)      ;; right subtree of t
  (node-key t)        ;; number labeling the root of t
  (insertavl t n)     ;; insert n to t, if not present
  (deleteavl t n)     ;; delete n from t, if present
  (listavl t)         ;; ordered list of elements in t
  (sizeavl t)         ;; the number of elements in t

  node-left, node-right, node-key, sizeavl: O(1)
  insertavl, deleteavl: O(log N)
  listavl: O(N)
</rkt>
```

*Implementation.* Set

```
<rkt>
  (require "avl-cs145.rkt")
  (provide emptyset emptyset? singleton union intersection difference size nth)

  ;; First, define a wrapper struct called "set"
  (define-struct set (data size))

  ;; Now, define basic functions
  (define emptyset (make-set empty 0))
  (define (emptyset? s) (empty? (set-data s)))
  (define (singleton n) (make-set (insertavl empty n) 1))
  (define (size s) (if (emptyset? s) 0 (set-size s)))

  ;; Union
  ;; O(N*log(M)), where N = (min (size s1) (size s2)), M = (max (size s1) (size s2))
  ;; Strategy: compare the sizes of two sets, then insert every elements from the smaller
  ;;           set to the larger set.
  ;; Running time: set traversal O(N); set insertion O(log(M)). Thus overall O(N*log(M)).

  ;; Consumes two trees, return a tree with elements from both trees
  ;; Recall that (insertavl t n) inserts n into t.
  ;; Pretty standard recursion.
  (define (union-helper t1 t2)
    (if (empty? t1)
      t2
      (insertavl
      (union-helper (node-left t1) (union-helper (node-right t1) t2)) (node-key t1))))

;; Consumes two sets, use union-helper to merge two trees
;; The program can be optimized by using local helper to compute union-helper only once and
;; thus runs faster but I'm too lazy to do it so I'll stick with the way I did my assignment.
(define (union s1 s2)
  (if (> (set-size s1) (set-size s2))
    (make-set (union-helper (set-data s2) (set-data s1))
              (sizeavl (union-helper (set-data s2) (set-data s1))))
    (make-set (union-helper (set-data s1) (set-data s2))
              (sizeavl (union-helper (set-data s1) (set-data s2))))))



;; Difference
;; Literally change insertavl to deleteavl and you are done.

(define (difference-helper t1 t2)
  (if (empty? t1)
    t2
    (deleteavl
      (difference-helper (node-left t1) (difference-helper (node-right t1) t2)) (node-key t1))))

(define (difference s2 s1)
  (make-set (difference-helper (set-data s1) (set-data s2)
            (sizeavl (difference-helper (set-data s1) (set-data s2))))))



;; Intersection
;; Property of set: s1 ∩ s2 = s1 \ (s1 \ s2)

(define (intersection s1 s2)
  (difference s1 (difference s1 s2)))



;; Finding ith smallest element in the set
;; Remark: since size starts at 1 but ith starts at 0 (e.g. if we are looking for the smallest
;;         node in the tree, we pass in i=0, but in the tree we are finding the "1st" node), to
;;         avoid confusion, when we pass in i to the helper, we add 1 to it.
;; Strategy:
;;   Case 1. When k <= size of left subtree, we know our target is in the left subtree, so
;;           we apply the function to the left subtree.
;;   Case 2. When k = size of left subtree plus 1, we return the root node. (This is actually the
;;           base case).
;;   Case 3. When k > size of left subtree, and the difference is more than one (so Case 2 fails),
;;           we apply the function to the right subtree. Note that k must subtract the size of
;;           left subtree, and also the root node (so k = k - left_size - 1)

(define (kth t k)
  (cond
    [(<= k (size-avl (node-left t))) (kth (node-left t) k)]
    [(= 1 (- k (sizeavl (node-left t)))) (node-key t)]
    [else (kth (node-right t) (sub1 (- k (sizeavl (node-left t)))))]))
  (define (nth s i)
    (kth (set-tree s) (add1 i)))
</rkt>
======================================================================================================================
```

END