Homework 3

Carl Eastlund

Due **Wed., Feb. 27** at **9:00pm**.

1. Associative Maps.

   (a) An `AssocMap` is a `Tree`:

   ```
   ;; A Tree is one of:
   ;; - empty
   ;; - (node Integer Key Value Tree Tree)
   (struct node [height key value left right] #:transparent)
   ;; where:
   ;; - for any node N in left, left.key < key
   ;; - for any node N in right, key < right.key
   ;; - height = 1 + max(left.height, right.height)
   ;; - left.height <= right.height + 1
   ;; - right.height <= left.height + 1

   ;; A Key is a Number
   ;; A Value is a String
   ```

   Note: Because the left and right subtrees of a node must have heights within 1 of each other, the maximum height of a tree is $O(\log n)$. The argument is based on the fact that we can construct the smallest possible tree for any height n. For height 0, the tree is `empty`. For height 1, the tree is a singleton node. For any higher height n, the smallest possible tree is a node with the smallest tree of height $n-1$ on the left and the smallest tree of height $n-1$ on the right. Adding the solutions for $n-1$ and $n-2$ suggest the Fibonacci numbers; we can verify that the smallest tree of height n has a number of leaves equal to one less than the $n+1$st Fibonacci number. Since the Fibonacci numbers grow exponentially, the minimum number of nodes for a given height therefore grows exponentially. The maximum height for a given number of nodes is the inverse of this function, and it therefore grows logarithmically. Interestingly, we get a logarithmic *height* bound without guaranteeing any constant factor bound for the relative *size* of the left and right subtrees.

   (b) i. Every key/value pair occupies a node, which takes $\Theta(1)$ space. Because keys are strictly increasing from left to right, there can be no duplicate keys, so there are n nodes for n keys. Therefore the total space is $\Theta(n)$.

   ii. `fresh-assoc` uses `empty-tree` which produces `empty`, which is trivially valid.

   iii. `assign` uses `tree-insert`. The `tree-insert` function recurs left or right depending on the given key, which preserves the order invariants on the tree. It builds nodes with `balanced-node` and `almost-balanced-node`, which automatically construct the height field correctly. Insertion can increase the height of a tree by at most one; `almost-balanced-node` preserves the balance invariants for trees that are off-balance by at most one by performing one or two tree rotations as needed. Therefore `tree-insert` produces a valid `Tree` at every step.

   iv. `unassign` uses `tree-delete-range` for a singleton range. The function `tree-delete-range` uses `tree-append`, `tree-filter<`, and `tree-filter>`. These functions all preserve the key order from the original tree. They all ultimately use `unbalanced-node`, `almost-balanced-node`, or balanced-node to construct nodes, which all maintain the height invariants. The unbalanced-node function recurs on the left or right to "push" the smaller subtree down inside the larger one to find siblings of similar size, then calls almost-balanced-node when it gets there. The result is always a balanced tree.

(c) i. We use worst-case analysis for our associative mappings.

ii. fresh-assoc produces empty immediately.

iii. assign calls tree-insert, which recurs $O(\log n)$ times and only calls $\Theta(1)$-time functions.

iv. unassign calls tree-append, tree-filter>, and tree-filter>. These functions all recur at most $O(\log n)$ times. They all also call unbalanced-node. The running time of unbalanced-node is odd; it runs in $O(|\log m - \log n|)$ time for trees of size m and n. Its running time is proportional to the *difference* between the heights of its arguments. As tree-append, tree-filter<, and tree-filter> recur, their calls to unbalanced-node "walk" down the height of a tree. If one call to unbalanced-node takes several steps by going from height $x+k$ to height x, the next call will start at height x. In other words, the *total* time spend in calls to unbalanced-node is $O(\log n)$ for any of tree-append, tree-filter<, and tree-filter>. Therefore all three functions take $O(\log n)$ time.

v. lookup calls tree-search which recurs at $O(\log n)$ times and calls no helpers.

2. (a) A Set is a Tree as defined above, in which every key represents an element of the set and the values associated with keys are irrelevant.

(b) i. Trees take $\Theta(n)$ space as argued above.

ii. empty-set calls empty-tree, which produces a valid tree as argued above.

iii. extend calls tree-insert, which produces a valid tree as argued above.

iv. without calls tree-delete-range, which produces a valid tree as argued above.

(c) i. Our bounds are all worst-case.

ii. empty-set takes $O(1)$ time trivially.

iii. in? calls tree-search, which takes $O(\log n)$ time as argued above.

iv. extend calls tree-insert, which takes $O(\log n)$ time as argued above.

v. without calls tree-delete-range, which takes $O(\log n)$ time as argued above.

3. (a)

```
;; A Queue is (queue Chain Chain).
(struct queue [leftmost rightmost] #:mutable #:transparent)
;; Either leftmost and rightmost are both empty, or they are both links.
;; If both are links, then they are part of a single chain of links:
;; * leftmost.left = empty
;; * rightmost.right = empty
;; * for any link L other than leftmost, L.left.right = L
;; * for any link L other than rightmost, L.right.left = L
;; * from any link L, leftmost is reachable by going left 0 or more times
;; * from any link L, rightmost is reachable by going right 0 or more times
;; These invariants must hold before and after any Queue operations.
;; Note that during Queue operations, in between mutations,
;; some or all of these invariants may be violated.

;; A Chain is either empty or (link Link String Link).
(struct link [left elem right] #:mutable #:transparent)
```

(b) i. A queue contains one link per value, and every link takes $\Theta(1)$ space. The total is therefore $\Theta(n)$ space.

ii. new-queue produces a queue that is trivially valid.

iii. add-leftmost maintains the left and right link invariants when adding links, and therefore produces a valid queue.

iv. `drop-leftmost` maintains the left and right link invariants when removing links and correctly detects when the queue becomes empty; it therefore produces a valid queue.

v. `add-rightmost` maintains the left and right link invariants when adding links, and therefore produces a valid queue.

vi. `drop-rightmost` maintains the left and right link invariants when removing links and correctly detects when the queue becomes empty; it therefore produces a valid queue.

vii. `concatenate` maintains the left and right link invariants, and ensures that every link is part of only one queue; it therefore produces a valid queue.

(c) The analysis for queues is worst-case, and by inspection every operation takes $\Theta(1)$ time because there is no recursion or iteration.