Homework 3

Carl Eastlund

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

Collaboration Policy: Your work on this assignment must be your own. You *may not* copy files from other students in this class, from people outside of the class, from the internet, or from any other source. You *may not* share files with other students in this class.

You *may* discuss the problems, concepts, and general techniques used in this assignment with other students, so long as you do not share actual solutions.

If you are in doubt about what you *may* and *may not* do, ask the course instructor before proceeding. If you violate the collaboration policy, you will receive a zero as your grade for this entire assignment and you will be reported to OSCCR (`northeastern.edu/osccr`).

You will implement three new datatypes in this assignment. Three are concrete datatypes; I will tell you what sort of representation to use, and you will implement that in Racket. Two are abstract datatypes; I will only tell you what operations they need to support, and you must both design the representation and implement your design in Racket.

For each datatype, you will be assigned a set of operations that the datatype must support, and an upper-bound on the running time of the operations. In each case, you will be asked to describe your design and analyze its efficiency in LaTeX (in `solution.tex` and `solution.pdf`) and implement it in Racket (in `solution.rkt`).

For implementing the datatypes, the *only* compound data structures you may use in Racket are lists, arrays, boxes, and `struct` definitions. The *only* built-in operations you may use on these data structures are those that run in $\Theta(1)$ time, plus `make-vector`, which runs in $\Theta(n)$ time. You must implement all other operations yourself. For reference, all the functions defined by `struct` are in $\Theta(1)$, as are `empty?`, `cons?`, `cons`, `first`, `rest`, `vector-length`, `vector-ref`, `vector-set!`, `box?`, `box`, and `unbox`.

In each case, any sequence of n operations must run in $O(n \log n)$ time in the average case. You can accomplish this using worst-case bounds as with our "D" heaps, or with amortized bounds as in our "growable sequences", or with average-case bounds as in hash tables.

In each case, a data structure containing n values must use at most $\Theta(n)$ space. Every `cons` cell, box, or `struct` takes up $\Theta(1)$ space plus the space for its contents; every vector of length n takes up n space plus the space for its contents. Numbers, strings, symbols, booleans, `empty` and `(void)` take up $\Theta(1)$ space each, for our purposes. (Technically, numbers and strings can be arbitrarily large, but we can consider only "small" numbers and strings for our purposes.)

1. Implement associative maps. In most representations, the `unassign` operation will be the most subtle, so bear it in mind during the design phase.

   `(fresh-assoc)` : *AssocMap*
   Creates a fresh associative map containing no associations.

   `(assign` *Number String AssocMap*`)` : *AssocMap*
   Adds an association that maps the given *Number* to the given *String*, overwriting any existing mapping for *Number*, returning the resulting *AssocMap*. Your implementation may construct a new *AssocMap* for the result, or mutate and return the original *AssocMap*.

   `(unassign` *Number AssocMap*`)` : *AssocMap*
   Removes any existing association for *Number*, returning the resulting *AssocMap*. Your implementation may construct a new *AssocMap* for the result, or mutate and return the original *AssocMap*.

`(lookup` *Number AssocMap*`)` : (*String* or `#false`)
Looks for an association for *Number*. If one exists, returns the corresponding *String*. Otherwise, returns `#false`.

(a) What is your data definition for *AssocMap*? Include all invariants necessary to achieve your asymptotic running time and space usage.

(b) Analyze the space used by your representation:

   i. Argue that any *AssocMap* matching your data definition that contains n associations uses at most $O(n)$ space.
   ii. Argue that `fresh-assoc` produces a valid *AssocMap*.
   iii. Argue that `assign` produces a valid *AssocMap*.
   iv. Argue that `unassign` produces a valid *AssocMap*.

(c) Analyze the running time for each operation:

   i. State whether your analysis is worst case, average case, or amortized.
   ii. Argue that `fresh-assoc` takes $O(1)$ time.
   iii. Argue that `assign` takes $O(\log n)$ time for an input that contains n associations.
   iv. Argue that `unassign` takes $O(\log n)$ time for an input that contains n associations.
   v. Argue that `lookup` takes $O(\log n)$ time for an input that contains n associations.

2. Implement sets. In most representations, the `without` operation will be the most subtle, so bear it in mind during the design phase.

   `(empty-set)` : *Set*
   Creates a set containing no elements.

   `(in?` *Number Set*`)` : *Boolean*
   Produces `#true` if the given *Set* contains the given *Number*; produces `#false` otherwise.

   `(extend` *Number Set*`)` : *Set*
   Produces a new set containing the given *Number* plus everything in the given *Set*. Your implementation may construct a new *Set*, or mutate and return the given *Set*.

   `(without` *Number Number Set*`)` : *Set*
   Produces a new set containing everything in the given *Set* except for elements between the two given *Number*s, inclusive.

3. Implement double-ended queues. In most representations, the `concatenate` operation will be the most subtle, so bear it in mind during the design phase.

   `(new-queue)` : *Queue*

   `(full?` *Queue*`)` : *Boolean*

   `(add-leftmost` *String Queue*`)` : *Queue*

   `(add-rightmost` *String Queue*`)` : *Queue*

   `(get-leftmost` *Queue*`)` : *String*

   `(get-rightmost` *Queue*`)` : *String*

   `(drop-leftmost` *Queue*`)` : *Queue*

   `(drop-rightmost` *Queue*`)` : *Queue*

   `(concatenate` *Queue Queue*`)` : *Queue*