Name: _____

CS4800: Algorithms and Data

Exam 2: Practice

Exam Policies. Your work on this exam must be your own. You *may not* copy solutions from other students. You *may not* communicate with other students during the exam. You *may not* communicate with anyone outside the class during the exam.

This exam is open book, open notes. You *may* refer to the text, prior homeworks, past exams, practice exams, your own notes, and anything else you brought with you on paper. You *may not* share notes or any other written materials with other students during the exam. You *may not* use electronic devices during the exam.

The contents of the exam are confidential. You *may not* discuss the contents of the exam with any student who has not yet taken the exam. Due to illness, travel, or other special circumstances, some students may take exams at different times than others.

You *must* complete the exam in the time provided. Your exam *will not* be accepted if you work past the deadline.

If you violate any of these rules, you will receive a zero as your grade for this exam and you will be reported to the Office of Student Conduct and Conflict Resolution (OSCCR).

In this exam, you will implement a sequence datatype represented as a *forest*, meaning a collection of trees.

Trees in our representation may be *leaves* or *branches*. Sequence elements are stored in the leaves of a tree; branches each have two subtrees but no elements of their own. We require branches to be perfectly balanced: both subtrees must have the same size.

Forests are represented as lists of trees of increasing size. It may be helpful to note that the sizes of the trees in a forest of arbitrary size n are closely related to the binary representation of the number n.

Our data definition is as follows:

```
;; A (Forest X) is a (Listof (Tree X)),
;; where the size of the trees is strictly increasing.

;; A (Tree X) is either:
;; - (leaf X), which is a tree of size 1.
;; - (branch Int (Tree X) (Tree X)),
;;   in which size is left.size + right.size,
;;   and left.size = right.size.

(struct leaf [value])
(struct branch [size left right])
```

Problem 1. Argue that the size of a tree must be a power of two.

Solution: We can prove this by induction. In the base case, a leaf has size $1 = 2^0$. For the inductive step, we consider branches. Based on the data definition for branches, the size of both subtrees must be the same. By our inductive hypothesis, the size of each subtree is 2^k for some natural number k. The size of the branch is therefore $2^k + 2^k = 2^{k+1}$.

Problem 2. Argue that a forest of size n contains $O(\log n)$ trees.

Solution: Because the size of every tree is a power of two, and the trees in a forest increase in size, the final tree in a forest with k trees must have at least 2^{k-1} elements. A forest containing more than $\lg n + 1$ trees would have to have at least one tree containing more than $2^{\lg n} = n$ elements, and would therefore have size greater than n. Therefore a forest of size n can contain at most $\lg n + 1 \in O(\log n)$ trees.

Problem 3. Implement the following functions to compute the size of forests and trees. State and justify the worst-case running time of both.

`(tree-size t)` : *Int*
`t` : `(Tree` X`)`
Reports the total number of leaves in `t`.

`(forest-size f)` : *Int*
`f` : `(Forest` X`)`
Reports the total number of leaves in all the trees in `f`.

Solution:

```
(define (forest-size f)
  (cond
    [(empty? f) 0]
    [(cons? f) (+ (tree-size (first f)) (forest-size (rest f)))]))

(define (tree-size t)
  (cond
    [(leaf? t) 1]
    [(branch? t) (branch-size t)]))
```

The running time of `tree-size` is $\Theta(1)$.
Since a forest contains $O(\log n)$ trees, `forest-size` calls `tree-size` $O(\log n)$ times. It therefore runs in $O(\log n)$ time.

Problem 4. Implement the following functions to look up elements of forests and trees by index. State and justify the worst-case running time of both.

`(tree-ref i t)` : X
`i` : *Int*
`t` : `(Tree` X`)`
Produces the `i`th element of `t`, indexed from 0. The size of `t` must be greater than `i`.

`(forest-ref i f)` : X
`i` : *Int*
`f` : `(Forest` X`)`
Produces the `i`th element of `f`, indexed from 0. The size of `f` must be greater than `i`.

Solution:

```
(define (forest-ref i f)
  (define t (first f))
  (define n (tree-size t))
  (cond
    [(< i n) (tree-ref i t)]
    [else (forest-ref (- i n) (rest f))]))

(define (tree-ref i t)
  (cond
    [(leaf? t) (leaf-value t)]
    [(branch? t)
     (define n (branch-size t))
     (cond
       [(< i (/ n 2)) (tree-ref i (branch-left t))]
       [else (tree-ref (- i (/ n 2)) (branch-right t))])]))
```

Because of the invariants on trees, the recurrence for the running time of `tree-ref` is $T(n) = T(n/2) + 1$. By the master method, that means `tree-ref` runs in $\Theta(\log n)$ time.

The function `forest-ref` recurs at most $O(\log n)$ times, doing a constant amount of work each time, before calling `tree-ref`. The total running time is therefore $O(\log n)$.

Problem 5. Implement the following functions to construct forests and trees. State and justify the worst-case running time of each.

`(tree-cons t f)` : `(Forest` X`)`
`t` : `(Tree` X`)`
`f` : `(Forest` X`)`
Adds `t` to the front of `f`. If `t` has the same size as the first tree in `f`, maintains the invariant that trees have increasing size by combining `t` with the first tree in `f` and recursively adding that to the forest.

`(forest-cons x f)` : `(Forest` X`)`
`x` : X
`f` : `(Forest` X`)`
Adds `x` to the front of `f`.

`(empty-forest)` : `(Forest` X`)`
Constructs an empty forest.

Solution:

```
(define (empty-forest) empty)

(define (forest-cons x f)
  (tree-cons (leaf x) f))

(define (tree-cons t f)
  (cond
    [(empty? f) (cons t empty)]
    [(cons? f)
     (define n (tree-size t))
     (cond
       [(< n (tree-size (first f))) (cons t f)]
       [else (tree-cons (branch (* 2 n) t (first f)) (rest f))])]))
```

Because a forest has $O(\log n)$ trees, `tree-cons` recurs at most $O(\log n)$ times. It does a constant amount of work at each step, so it runs in $O(\log n)$ time.

The function `forest-cons` simply calls `leaf` and `tree-cons`, so its running time is also $O(\log n)$.

The running time of `empty-forest` is $\Theta(1)$.

Problem 6. Implement the following functions to remove elements from forests and trees. State and justify the worst-case running time of both.

`(tree-rest t f)` : `(Forest` X`)`
`t` : `(Tree` X`)`
`f` : `(Forest` X`)`
The tree `t` must be smaller than every tree in `f`. Adds every element *but* the first in `t` to `f` by adding all the right subtrees of `t` to `f`, and recurring on the left.

`(forest-rest f)` : `(Forest` X`)`
`f` : `(Forest` X`)`
Produces a forest containing every element in `f` but the first.

Solution:

```
(define (forest-rest f)
  (tree-rest (first f) (rest f)))

(define (tree-rest t f)
  (cond
    [(leaf? t) f]
    [(branch? t)
     (tree-rest (branch-left t)
       (cons (branch-right t) f))]))
```

Because our trees are perfectly balanced, the recurrence for the running time of `tree-rest` in terms of the size of `t` is $T(n) = T(n/2) + 1$. By the master method, this is $\Theta(\log n)$.

Since `forest-rest` simply calls `tree-rest` on its smallest tree, it runs in $O(\log n)$ time.

Extra Credit. Argue that if we consider only the operations `empty-forest`, `forest-cons`, `forest-size`, and `forest-ref`, the *amortized* cost of `forest-cons` is $\Theta(1)$.