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).

Problem A. In this problem, you will build an implementation for 2-key dictionaries from 1-key dictionaries. Assume you are given a representation for 1-key dictionaries and implementations for their operations as described below:

`(Dict` K V`)`
: This datatype represents a dictionary mapping keys of type K to values of type V. Works for any given types K and V. A dictionary with n keys takes $\Theta(n)$ space, plus the space used by the keys and values.

`(empty-dict)` : `(Dict` K V`)`
: Produces an empty dictionary in $\Theta(1)$ time.

`(is-dict-empty? d)` : *Bool*
: `d` : `(Dict` K V`)`
: Returns `#true` if `d` is empty, or `#false` otherwise. Runs in $\Theta(1)$ time.

`(find-in-dict k d)` : V or `#false`
: `k` : K
: `d` : `(Dict` K V`)`
: If `d` maps `k` to `v`, returns `v`; returns `#false` otherwise. Runs in $\Theta(\log n)$ time, where `d` has n keys.

`(add-to-dict k v d)` : `(Dict` K V`)`
: `k` : K
: `v` : V
: `d` : `(Dict` K V`)`
: Produces an updated version of `d` that maps `k` to `v`. Runs in $\Theta(\log n)$ time, where `d` has n keys.

`(remove-from-dict k d)` : `(Dict` K V`)`
: `k` : K
: `d` : `(Dict` K V`)`
: Produces an updated version of `d` with no mapping for `k`. Runs in $\Theta(\log n)$ time, where `d` has n keys.

Example. Here we adapt 1-key dictionaries to represent and implement finite sets supporting the operations `empty-set`, `is-set-empty?`, `exists-in-set?`, `add-to-set`, and `remove-from-set`.

A `(Set` X`)` is a `(Dict` X `#true)`. Every value in the set is mapped to `#true` in its representation as a dictionary. Any value without a mapping is not in the set.

```
;; empty-set : -> (Set X)
;; Produces an empty set, represented as an empty dictionary.
(define (empty-set) (empty-dict))

;; is-set-empty? : (Set X) -> Bool
;; Reports whether s is empty using is-dict-empty?.
(define (is-set-empty? s) (is-dict-empty? s))

;; exists-in-set? : X (Set X) -> Bool
;; Reports whether x is found in s by using find-in-dict.
(define (exists-in-set? x s) (find-in-dict x s))

;; add-to-set : X (Set X) -> (Set X)
;; Produces an updated version of s that contains x.
(define (add-to-set x s) (add-to-dict x #true s))

;; remove-from-set : X (Set X) -> (Set X)
;; Produces an updated version of s that does not contain x.
(define (remove-from-set x s) (remove-from-dict x s))
```

Note that the adaptation defines `(Set` X`)` in terms of `(Dict` K V`)`, instantiates the types K and V for its own purposes, and explains what the keys and values in the dictionary represent when used as a set.

Requirements. You must provide definitions for the following types and operations:

`(PairDict` A B C`)`
: This datatype represents a "pair-dictionary" that maps pairs of keys, one of type A and one of type B, to values of type C. Works for any given types A, B, and C. A pair-dictionary with n pairs of keys takes $\Theta(n)$ space, plus the space used by the keys and values themselves.

`(empty-pair-dict)` : `(PairDict` A B C`)`
: Produces an empty pair-dictionary in $\Theta(1)$ time.

`(is-pair-dict-empty? d)` : *Bool*
: `d` : `(PairDict` A B C`)`
Returns `#true` if `d` is empty, or `#false` otherwise. Runs in $\Theta(1)$ time.

`(find-in-pair-dict a b d)` : C or `#false`
: `a` : A
`b` : B
`d` : `(PairDict` A B C`)`
If `d` maps the pair of `a` and `b` to `c`, returns `c`; returns `#false` otherwise. Runs in $\Theta(\log n)$ time, where `d` has n pairs of keys.

`(add-to-pair-dict a b c d)` : `(PairDict` A B C`)`
: `a` : A
`b` : B
`c` : C
`d` : `(PairDict` A B C`)`
Produces an updated version of `d` that maps the pair of `a` and `b` to `c`. Runs in $\Theta(\log n)$ time, where `d` has n pairs of keys.

`(remove-from-pair-dict a b d)` : `(PairDict` A B C`)`
: `a` : A
`b` : B
`d` : `(PairDict` A B C`)`
Produces an updated version of `d` with no mapping for the pair of `a` and `b`. Runs in $\Theta(\log n)$ time, where `d` has n pairs of keys.

A.1 Write a data definition for `(PairDict` A B C`)`. Argue that a pair-dictionary with n pairs of keys uses $\Theta(n)$ space, in addition to the space used by the keys and values themselves.

Hint: think "nested dictionaries".

Solution: A `(PairDict` A B C`)` is a `(Dict` A `(Dict` B C`))`, in which each `(Dict` B C`)` must be non-empty. Every mapping from the pair of keys `a` and `b` to some value `c` is represented by a mapping from `a` to `d` in the outer dictionary, where `d` is an inner dictionary that maps `b` to `c`.

Every key `a` in the outer dictionary corresponds to the first element of one or more of the n key pairs in a pair-dictionary. (If the outer dictionary could contain empty inner-dictionaries, it would be possible to have more than n outer keys, and our space invariant would be broken.) Every key `b` in an inner dictionary corresponds to the second element of exactly one of the n key pairs in a pair-dictionary. Since the total number of dictionary keys is therefore between n and $2n$, the total space used by the pair-dictionary is $\Theta(n)$.

A.2 Implement `empty-pair-dict`. Argue that it runs in $\Theta(1)$ time.

Solution:

```
(define (empty-pair-dict) (empty-dict))
```

This must run in $\Theta(1)$ time because `empty-dict` does.

A.3 Implement `is-pair-dict-empty?`. Argue that it runs in $\Theta(1)$ time.

Solution:

```
(define (is-pair-dict-empty? d) (is-dict-empty? d))
```

This must run in $\Theta(1)$ time because `is-dict-empty?` does.

Note that the result of `is-dict-empty?` is correct because a pair-dictionary cannot contain empty inner dictionaries.

A.4 Implement `find-in-pair-dict`. Argue that it runs in $O(\log n)$ time.

Solution:

```
(define (find-in-pair-dict a b d)
  (find-in-dict b (find-nested-dict a d)))

(define (find-nested-dict a d)
  (define d2 (find-in-dict a d))
  (cond
    [(equal? d2 #false) (empty-dict)]
    [else d2]))
```

This must run in $O(\log n)$ time because both outer and inner dictionaries must have $O(n)$ keys, and `find-in-dict` runs in $O(\log n)$ time. The helper `find-nested-dict` runs in $O(\log n)$ time for the same reason.

A.5 Implement `add-to-pair-dict`. Argue that it runs in $O(\log n)$ time.

Solution:

```
(define (add-to-pair-dict a b c d)
  (add-to-dict a (add-to-dict b c (find-nested-dict a d)) d))
```

This must run in $O(\log n)$ time because both outer and inner dictionaries must have $O(n)$ keys, and both `find-nested-dict` and `add-to-dict` run in $O(\log n)$ time.

A.6 Implement `remove-from-pair-dict`. Argue that it runs in $O(\log n)$ time.

Solution:

```
(define (remove-from-pair-dict a b d)
  (add-nested-dict a (remove-from-dict b (find-nested-dict a d)) d))

(define (add-nested-dict a d2 d)
  (cond
    [(is-dict-empty? d2) (remove-from-dict a d)]
    [else (add-to-dict a d2 d)]))
```

This must run in $O(\log n)$ time because both outer and inner dictionaries must have $O(n)$ keys, and the functions `find-in-dict`, `add-to-dict`, and `remove-from-dict` run in $O(\log n)$ time. The helper `add-nested-dict` runs in $O(\log n)$ time as well, for the same reason.

Note that we need `add-nested-dict` to preserve the invariant that a pair-dictionary contains only non-empty inner dictionaries.

Problem B. In this problem, you will implement a *priority queue*. A priority queue is like a heap, in that it provides fast access to the item with the lowest priorty value. In addition, a priority queue supports an operation to change the priority of an existing element.

`(PrioQ T)`

A `(PrioQ T)` is a priority queue containing elements of type T with integer priorities. We represent priority queues as *binary heaps*.

A binary heap is *conceptually* represented as a binary tree in which every node has a value and a priority queue. The shape of the tree—the positions of its nodes and leaves—is determined uniquely by its size. Essentially, nodes in the tree are filled in from left to right at the bottom depth of the tree; when one depth is full, the next node goes at the next depth.

Because of the strict restrictions on the shape of a binary heap, it can be *concretely* represented as a vector. The root of the tree is the element at index 0. From any node at index i, its left child is found at index $2i$, its right child is at index $2i+1$, and its parent node is found at index $\lfloor i/2 \rfloor$. A binary heap with n elements occupies indices 0 to $n-1$ of a vector.

We will assume we do not know the maximum size of our priority queue when we create it. We must therefore dynamically grow and shrink the vector as with growable sequences and hash tables.

Our data definition is as follows:

```
;; A (PrioQ T) is (pq Int (Vectorof (Or (Node T) #false)))
;; where:
;; - (vector-length vector) is always a power of 2
;; - size is at least (quotient (vector-length vector) 4)
;; - size is at most (vector-length vector)
;; - elements 0 to size-1 of vector are nodes
;; - all other elements of vector are #false
;; - every node's index field is the same as its index in the vector
;; - every node's priority is greater than or equal to its parent's priority
(struct pq [size vector] #:mutable)

;; A (Node T) is (node Int Int T)
(struct node [index priority value] #:mutable)
```

B.1 Implement the following function:

`(empty-pq)` : `(PrioQ` *T*`)`
Produces an empty priority queue.

Solution:

```
(define (empty-pq)
  (pq 0 (make-vector 1 #false)))
```

B.2 Implement the following function:

`(is-pq-empty? q)` : *Bool*
`q` : `(PrioQ` *T*`)` Reports whether `q` is empty in $\Theta(1)$ time.

Solution:

```
(define (is-pq-empty? q)
  (zero? (pq-size q)))
```

B.3 Implement the following three functions:

`(parent n q)` : `(Node` T`)` or `#false`

`(left-child n q)` : `(Node` T`)` or `#false`

`(right-child n q)` : `(Node` T`)` or `#false`
`n` : `(Node` T`)`
`q` : `(PrioQ` T`)`
Given a node and the priority queue that contains it, returns the parent, left child, or right child of the node, respectively. Returns `#false` if the node does not have a parent, left child, or right child, respectively, within the size of the priority queue. All three functions must run in $\Theta(1)$ time.

Solution:

```
(define (parent n q)
  (define i (node-index n))
  (cond
    [(zero? i) #false]
    [else (vector-ref (pq-vector q) (quotient i 2))]))

(define (left-child n q)
  (define i (* 2 (node-index n)))
  (cond
    [(>= i (pq-size q)) #false]
    [else (vector-ref (pq-vector q) i)]))

(define (right-child n q)
  (define i (add1 (* 2 (node-index n))))
  (cond
    [(>= i (pq-size q)) #false]
    [else (vector-ref (pq-vector q) i)]))
```

B.4 Implement the following function:

`(shrink! q)` : *Void*
`q` : `(PrioQ T)`
If `q` is using one quarter or less of its allocated vector's size, and its size can be cut in half, copies the contents of `q` to a vector of half the size. Otherwise, does nothing. Runs in $\Theta(n)$ time when `q` needs to be resize, and $\Theta(1)$ time otherwise.

Solution:

```
(define (shrink! q)
  (define n (pq-size q))
  (define v (pq-vector q))
  (define v.n (vector-length v))
  (cond
    [(or (< v.n 2) (> n (quotient v.n 4)))
     (void)]
    [else
     (define v2.n (quotient v.n 2))
     (define v2 (make-vector v2.n #false))
     (vector-copy! v v2 v2.n)
     (set-pq-vector! pq v2)]))

(define (vector-copy! v1 v2 n)
  (cond
    [(zero? n) (void)]
    [else
     (define i (sub1 n))
     (vector-set! v2 i (vector-ref v1 i))
     (vector-copy! v1 v2 i)]))
```

B.5 Implement the following function:

`(grow! q)` : *Void*
`q` : `(PrioQ` *T*`)`
If `q` is using all of its allocated vector's size, copies the contents of `q` to a vector of twice the size. Otherwise, does nothing. Runs in $\Theta(n)$ time when `q` needs to be resize, and $\Theta(1)$ time otherwise.

Solution:

```
(define (grow! q)
  (define n (pq-size q))
  (define v (pq-vector q))
  (define v.n (vector-length v))
  (cond
    [(< n v.n)
     (void)]
    [else
     (define v2.n (* v.n 2))
     (define v2 (make-vector v2.n #false))
     (vector-copy! v v2 v2.n)
     (set-pq-vector! pq v2)]))
```

B.6 While implementing binary heaps, we will temporarily violate the property that every node's priority is greater than or equal to its parent's priority. The property will always hold everywhere but one node—the node we are currently working on may have a priority lower than its parent, or higher than one or both of its children. Implement the following function to help restore a "broken" tree:

`(node-to-fix n q)` : `(Node` T`)` or `#false`
`n` : `(Node` T`)`
`q` : `(PrioQ` T`)`
If `n` has a parent node in `q` and `n`'s priority is less than its parent's priority, produces the parent node.
Otherwise, if `n` has any children and `n`'s priority is greater than one or both of its children's priorities, produces the child of `n` that has the lowest priority.
Finally, produces `#false` if neither condition above holds.
Runs in $\Theta(1)$ time.

Solution:

```
(define (node-to-fix n q)
  (define p (parent n q))
  (define l (left-child n q))
  (define r (right-child n q))
  (cond
    [(and (node? p)
       (< (node-priority n) (node-priority p)))
     p]
    [(and (node? l)
       (> (node-priority n) (node-priority l))
       (or (not (node? r))
         (<= (node-priority l) (node-priority r))))
     l]
    [(and (node? r)
       (> (node-priority n) (node-priority r)))
     r]
    [else #false]))
```

B.7 Given two nodes that are out of order, we must begin to fix the tree by swapping those two nodes. Implement the following function:

`(swap-nodes! n1 n2 q)` : *Void*
`n1` : `(Node` T`)`
`n2` : `(Node` T`)`
`q` : `(PrioQ` T`)`
Swaps `n1` and `n2` in `q` by swapping them in `q`'s vector and updating the indices stored in the nodes. Runs in $\Theta(1)$ time.

Solution:

```
(define (swap-nodes! n1 n2 q)
  (define i1 (node-index n1))
  (define i2 (node-index n2))
  (define v (pq-vector q))
  (vector-set! v i1 n2)
  (vector-set! v i2 n1)
  (set-node-index! n1 i2)
  (set-node-index! n2 i1))
```

B.8 When we "break" some node in a binary heap, we must repeatedly swap that node with one of its neighbors, "following" the original node until it is in the right place in the tree. Implement the following function, and be sure to use `node-to-fix` and `swap-nodes!` as appropriate:

`(fix! n q)` : *Void*
`n` : `(Node` *T*`)`
`q` : `(PrioQ` *T*`)`
Swaps `n` with one of its neighbors, following it up or down a binary heap until it finds its proper place. Runs in $O(\log n)$ time.

Solution:

```
(define (swap-nodes! n1 n2 q)
  (define i1 (node-index n1))
  (define i2 (node-index n2))
  (define v (pq-vector q))
  (vector-set! v i1 n2)
  (vector-set! v i2 n1)
  (set-node-index! n1 i2)
  (set-node-index! n2 i1))
```

B.9 Argue that once `fix!` moves `n` downward, `n` will not need to immediately swap upward with its parent.

Solution: If `n` moves downward, then it has already been compared with its new parent, and belongs below it. Swapping it back upward would merely restore the previous "broken" state.

B.10 Argue that once `fix!` moves `n` upward, `n` will not need to immediately swap downward with either of its children.

Solution: If `n` moves upward, then its priority was less than that of its parent. Given that `n`'s parent and `n`'s sibling were correctly ordered, `n` is guaranteed to have a lower priority than both of its children in its new position. Therefore `n` will not need to swap downward.

B.11 Argue that in a single call to `fix!`, either `n` moves only upward or `n` moves only downward; `n` never alternates moving up and down.

Solution: Based on the previous two arguments, there can be no two swaps where `n` switches from moving up to moving down. Therefore, once `n` moves in either direction, it cannot turn back.

B.12 Argue that `fix!` runs in $O(\log n)$ time.

Solution: If `n` only moves up or down, the number of swaps is bounded by the height of the tree, which is $\lg n$. If each swap takes $\Theta(1)$ time, the total therefore takes $O(\log n)$ time.

B.13 We add elements to a priority queue by adding a new node at the bottom of the tree, then fixing up the tree by allowing the node to "bubble up" to its proper position. Implement the following function:

`(pq-add! x p q)` : `(Node` T`)`
`x` : T
`p` : *Int*
`q` : `(PrioQ` T`)`
Adds a value `x` with priority `p` to priority queue `q`. Returns the resulting node containing `v`. Runs in $O(\log n)$ time.

Solution:

```
(define (pq-add! x p q)
  (grow! q)
  (define i (pq-size q))
  (define v (pq-vector q))
  (define n (node i p x))
  (set-pq-size! q (add1 i))
  (vector-set! v i n)
  (fix! n q)
  n)
```

B.14 To update the priority of an element, we simply change the priority of its node and then allow that node to "bubble up" or "bubble down" to its proper position. Implement the following function:

`(pq-update! n p q)` : *Void*
`n` : `(Node` T`)`
`p` : *Int*
`q` : `(PrioQ` T`)`
Updates the priorty of the value in `n` to be `p` in `q`. Runs in $O(\log n)$ time.

Solution:

```
(define (pq-update! n p q)
  (set-node-priority! n p)
  (fix! n q))
```

B.15 The least element of a binary heap is always at the root of the tree. We remove the root by swapping the root with a leaf node, removing the leaf, and then allowing the new root to "bubble down" to its proper position. Implement the following function:

`(pq-remove! q)` : T
`q` : `(PrioQ` T`)`
Returns the value in `q` with the lowest priority. Removes that value from the queue. Runs in $O(\log n)$ time.

Solution:

```
(define (pq-remove! x p q)
  (shrink! q)
  (define i (sub1 (pq-size q)))
  (define v (pq-vector q))
  (define r (vector-ref v 0))
  (define n (vector-ref v i))
  (swap-nodes! r n q)
  (set-pq-size! q i)
  (fix! n q)
  (node-value r))
```