Fall CS 145 Final Review Guide.

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

==

Topic. 3: Tree

Subtopic. 3.1: Basics about tree

3.1.1 *Implementation.*

```
<rkt>
  (define-struct node (left right))

  ;; (make-node l r)
  ;; (node-left t)
  ;; (node-right t)
  ;; (node? x)

</rkt>
```

3.1.2 *Note.*

1. Path: sequence of nodes leading from root to another node.
2. Depth: number of nodes in the path from root to node.
3. Height: max depth of all nodes in the tree.
4. Leaf: nodes at the bottom which have no children.

3.1.3 *Note.*

1. Given height h, the maximum number of nodes: 2^h-1; the minimum number of nodes: h.
2. Given number of elements n, the maximal height for a bst: n
   the minimal&average height for a bst: log n
3. Given number of elements n, the number of maximal height binary trees: 2^(n-1)
   the number of minimal height binary trees:
4. Running time
   a. Time to insert into a bst with n elements, worst case: O(n).
   b. Time to insert into a bst with h height, worst case: O(h).
   c. Height of AVL tree with n elements, worst case: O(log n).
   d. Time to append two Racket lists, where n is the sum of lists, worst case: O(n).
   e. Time to sort n elements, best algorithm: O(n*log(n)).

Subtopic. 3.2 Operations on general binary trees

Remark.
Contains #A3b~d

Implementation. Helper: is the current-node leaf?

```
<rkt>
  (define (is-leaf? t)
    (and (not (empty? t)
         (and (empty? (node-left t)) (empty? (node-right t))))))
</rkt>
```

Implementation. Count how many leaves t has.

<rkt>

```
  (define (count-leaves t)
    (cond
      [(empty? t) 0]
      [(is-leaf? t) 1]
      [else (+ (count-leaves (node-left t)) (count-leaves (node-right t)))]))
```

</rkt>

Implementation. Cut all leaves of t.

<rkt>

```
  (define (prune t)
    (cond
      [(empty? t) empty]
      [(is-leaf? t) empty]
      [else (make-node (prune (node-left t)) (prune (node-right t)))]))
```

</rkt>

Implementation. Cut kth leaf of t.

<rkt>

```
  (define (prune-kth t k)
    (cond
      [(is-leaf? t) empty]
      [(< k (count-leaves (node-left t))) (make-node (prune-kth (node-left t) k) (node-right t))]
      [else (make-node (node-left t) (prune-kth (node-right t) (- k (count-leaves (node-left t)))))]))
```

</rkt>

<stepper>

```
First, define a tree for testing purpose.

~> (define tree
     (make-node
       (make-node
         (make-node empty empty)
         (make-node empty empty))
       (make-node
         (make-node empty empty)
         (make-node empty empty))))

~> (prune tree 1) ;; get rid of the 1+1=2nd leaf.

We want:
     (make-node
       (make-node
         (make-node empty empty)
         empty
       (make-node
         (make-node empty empty)
         (make-node empty empty))))
```

```
Start stepper:
~> (count-leaves (node-left t)) = 2; k = 1; thus the second clause is executed.
=> (make-node (prune-kth (node-left tree 1)) (node-right t))

~> (count-leaves (node-left t)) = 1; k = 1; thus the third clause is executed.
=> (make-node (node-left t) (prune-kth (node-right t) (- 1 1)))

~> Now t is a leaf node, thus the first clause is executed.
~> Since we have recursion, the entire process is finished. We now have a new tree without
   the second leaf node.
```
</stepper>

Implementation. Measuring the height of t

<rkt>
```
  (define (height t)
    (cond
      [(empty? t) 0]
      [(is-leaf? t) 1]
      [else (add1 (max (height (node-left t)) (height (node-right t))))]))
```
</rkt>

Implementation. Create a min tree with n nodes.

Note. Two cases:

1. n = odd: for example, let n = 9. Ignore the root node we have n = 8.
   Apply n = 4 to both subtrees.
2. n = even: for example, let n = 8. Ignore the root node we have n = 7.
   Apply n = 4 to the left and n = 3 to the right (You can do either way).

<rkt>
```
  (define (tree-create-min n)
    (cond
      [(zero? n) empty]
      [(= 1 n) (make-node empty empty)]
      [else
        (let* {[new-n (quotient (sub1 n) 2)]}
          (cond
            [(odd? n) (make-node
                        (tree-create-min new-n)
                        (tree-create-min new-n))]
            [else (make-node
                      (tree-create-min new-n)
                      (tree-create-min (add1 new-n)))])))]))
```
</rkt>

Implementation. Create a min tree with one extra node

<rkt>
```
  (define (count-node t)
    (cond
      [(empty? t) 0]
      [(is-leaf? t) 1]
```

```
      [else (add1 (+ (count-node (node-left t)) (count-node (node-right t))))

  (define (tree-grow-min t)
    (cond
      [(empty? t) (make-node empty empty)]
      [(< (count-node (node-left t)) (count-node (node-right t)))
       (make-node (tree-grow-min (node-left t)) (node-right t))]
      [else (make-node (node-left t) (tree-grow-min (node-right t)))]))
</rkt>
```

Implementation. Create a min tree with one less node

```
<rkt>
  (define (tree-shrink-min t)

    ;; if the tree has 2^k nodes then we build a new tree.
    (define (is-special? t)
      (= (expt 2 (height t)) (count-nodes t)))

    ;; the magical function allows us to build a heigh n tree with n applications of make-node
    (define (echo t) (make-node t t))
    (define (build-tree n)
      (if (zero? n) empty (echo (build-tree (sub1 n)))))

    (cond
      [(= 1 (count-node t)) empty]
      [(is-speical? t) (build-tree (sub1 (height t)))]
      [else
        (if (empty? (node-left t))
          (make-node (node-left t) (tree-shrink-min (node-right t)))
          (make-node (tree-shrink-min (node-left t)) (node-right t)))]))
</rkt>
```

Subtopic. 3.3 Annotated binary tree (#A4)

Implementation. Struct

```
<rkt>
  (define-struct cnode (left right value))  ;; now each node has an extra buildin field
</rkt>
```

Implementation. Fast-leaf-count

```
<rkt>
  (define-struct cnode (left right leaf))
  (define (fast-leaf-count t)
    (if (empty? t) 0 (cnode-leaf t)))
</rkt>
```

Implementation. Tree-grow-min with extra field

```
<rkt>
  ;; size measures the size of the tree
  (define-struct node (left right size))
```

```
  ;; count how many node a tree has; uses if statement to catch exception
  (define (fast-count t)
    (if (empty? t) 0 (node-size t)))

  (define (tree-grow-min t)
    (cond
      [(empty? t) (make-node empty empty 1)]
      [(= (node-size t) 1) (make-node (make-node empty empty 1) empty 2)]
      [(and (empty? (node-left t)) (not (empty? (node-right t))))
       (make-node (tree-grow-min (node-left t)) (node-right t) (+ 1 (node-size t)))]
      [(and (empty? (node-right t)) (not (empty? (node-left t))))
       (make-node (node-left t) (tree-grow-min (node-right t)) (+ 1 (node-size t)))]
      [(< (node-size (node-left t)) (node-size (node-right t)))
       (make-node (tree-grow-min (node-left t)) (node-right t) (+ 1 (node-size t)))]
      [else
       (make-node (node-left t) (tree-grow-min (node-right t)) (+ 1 (node-size t)))]))
</rkt>
```

Implementation. Tree-shrink-min with extra field

```
<rkt>
  ;; check if a tree is perfect
  (define (is-perfect? t)
    (if (empty? t) true (integer? (/ (log (add1 (node-size t))) (log 2)))))

  ;; check if the tree is "special", aka has 2^k nodes.
  (define (is-special? t)
    (integer? (/ (log (node-size t)) (log 2))))

  ;; directly calculate the height of the tree
  (define (height t)
    (inexact->exact (/ (log (node-size t)) (log 2))))

  ;; see echo function above
  (define (echo t)
    (if (not (empty? t))
      (make-node t t (+ 1 (* 2 (node-size t))))
      (make-node t t 1)))

  ;; see build-tree above
  ;; but now we are working with height
  (define (build-tree h)
    (if (zero? h) empty (echo (build-tree (sub1 h)))))

  (define (tree-shrink-min t)
    (cond
      [(empty? t) empty]
      [(= 1 (node-size t)) empty]
      [(is-perfect? t)
       (cond
         [(is-leaf? (node-left t)) (make-node empty (node-right t) (sub1 (node-size t)))]
         [else (make-node (tree-shrink-min (node-left t)) (node-right t) (sub1 (node-size t)))])]
      [(is-special? t) (build-tree (height t))]
      [(> (node-size (node-left t)) (node-size (node-right t)))
```

```
       (make-node (tree-shrink-min (node-left t)) (node-right t) (sub1 (node-size t)))]
      [else
       (make-node (node-left t) (tree-shrink-min (node-right t)) (sub1 (node-size t)))]))
</rkt>

-----------------------------------------------------------------------------------------------------------------------------------
```

Subtopic. 3.4 BST (#A5)

Implementation. Struct

```
<rkt>
  (define-struct node (left right key))
</rkt>
```

Implementation. Sum

```
<rkt>
  (define (sum bst)
    (cond
      [(empty? bst) 0]
      [(is-leaf? bst) (node-key bst)]
      [else (+ (node-key bst) (sum (node-left bst)) (sum (node-right bst)))]))
</rkt>
```

Implementation. Member

```
<rkt>
  (define (member bst e)
    (cond
      [(empty? bst) false]
      [(= e (node-key bst)) true]
      [else (or (member (node-left bst) e) (member (node-right bst) e))]))
</rkt>
```

Implementation. Insert

```
<rkt>
  (define (insert bst e)
    (cond
      [(empty? bst) (make-node empty empty e)]
      [(= e (node-key bst)) bst]
      [(< e (node-key bst)) (make-node (insert (node-left bst) e) (node-right bst) (node-key bst))]
      [else (make-node (node-left bst) (insert (node-right bst) e) (node-key bst))]))
</rkt>
```

Implementation. Delete

```
<rkt>
  (define (single-left? t)
    (and (not (is-leaf? t))
         (not (empty? (node-left t)))
         (empty? (node-right t))))

  (define (single-right? t)
    (and (not (is-leaf? t))
```

```
        (not (empty? (node-right t)))
        (empty? (node-left t))))

  (define (find-right-most t) ;; returns the key which will serve as our replacement
    (if (empty? (node-right t)) (node-key t) (find-right-most (node-right t))))

  (define (delete bst e)
    (cond
      [(empty? bst) empty]
      [(= e (node-key bst))
       (cond
        [(single-left? bst)
         (make-node (node-left (node-left bst)) (node-right (node-left bst)) (node-key (node-left bst)))]
        [(single-right? bst)
         (make-node (node-left (node-right bst)) (node-right (node-right bst)) (node-key (node-right bst)))]
        [else
         (make-node (delete (node-left bst) (find-right-most (node-left bst)))
                  (node-right bst)
                  (find-right most (node-left bst)))])]
      [(< e (node-key bst)) (make-node (delete (node-left bst) e) (node-right bst) (node-key bst))]
      [else (make-node (node-left bst) (delete (node-right bst) e) (node-key bst))]))

</rkt>
```

Implementation. Combine

```
<rkt>
  (define (tree->lst bst)
    (if (empty? bst)
      empty
      (append (tree->lst (node-left bst)) (cons (node-key bst) (tree->lst (node-right bst))))))

  (define (insertlist lst t)
    (if (empty? lst)
      t
      (insert (insertlist (rest lst) t) (first lst))))

  (define (combine bst1 bst2)
    (if (< (node-size bst1) (node-size bst2))
      (insertlist (tree->lst bst1) bst2)
      (insertlist (tree->lst bst2) bst1)))
</rkt>
```

Subtopic. 3.5 BST <=> List

Implementation. bst->list, ascending order

```
<rkt>
  (define (bst->list bst)
    (define (helper bst acc)
      (if (empty? bst) acc
        (helper (node-left bst) (cons (node-key bst) (helper (node-right bst)))))
    (helper bst '()))
</rkt>
```

Implementation. list->bst

<rkt>

```
  (define (list->bst lst)
    (define (helper lst acc)
      (if (empty? lst) acc
        (helper (rest lst) (insert acc (first lst)))))
    (helper lst '()))
```

</rkt>

Implementation. ordered-list->balanced-bst

<rkt>

```
  (define (take lst n)
    (define (helper lst n acc)
      (if (zero? n) acc
        (cons (first lst) (helper (rest lst) (sub1 n) acc))))
    (helper lst n '()))

  (define (drop lst n)
    (if (zero? n) lst (drop (rest lst) (sub1 n))))

  (define (left-subtree lst)
    (take lst (quotient (length lst) 2)))

  (define (root lst)
    (first (drop lst (quotient (length lst) 2))))

  (define (right-subtree lst)
    (rest (drop lst) (quotient (length lst) 2)))

  (define (ordered->balanced lst)
    (if (empty? lst) empty
      (make-node (ordered->balanced (left-subtree lst))
                 (ordered->balanced (right-subtree lst))
                 (root lst))))
```

</rkt>

Implementation. unordered-list->balanced-bst

<rkt>

```
  (define (unordered->bst lst)
    (ordered-balanced (bst->list (list->bst lst))))
```

</rkt>

==

END

[redacted]