**Structures and Interpretation of Computer Program**

**Exercise Chapter 2.3 Name:** Wan Huzaifah bin Wan Azhar

**Exercise 2.3.3 Example: Representing Sets**

(define (element-of-set? x set)

(cond ((null? set) #f)

((equal? x (car set)) #t)

(else (element-of-set? x (cdr set)))))

(define (union-set set1 set2)

(cond

((null? set1) set2)

((not (element-of-set? (car set1) set2))

(union-set (cdr set1) (cons (car set1) set2)))

(else (union-set (cdr set1) set2))))

(define test-set1 '(a b c d n r))

(define test-set2 '(a b c d x z))

(display (union-set test-set1 test-set2))



(define (element-of-set? x set)

(cond ((null? set) #f)

((equal? x (car set)) #t)

(else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)

(cons x set))

(define (intersection-set set1 set2)

;stays the same as textbook

)

(define (union-set set1 set2)

(append set1 set2))

(define test-set1 '(a b c d n r))

(define test-set2 '(a b c d x z))

(display (union-set test-set1 test-set2))

* Efficiency of non-duplicate vs duplicate:
  + O(1) for adjoin-set duplicate vs O(n) for non-duplicate.
  + Intersection stays the same O(n2)
  + O(n) for union-set (as append is O(n)) for duplicate vs O(n2) for non-duplicate



(define (adjoin-set x set)

(cond ((null? set) (cons x '()))

((= x (car set)) set)

((< x (car set)) (cons x set))

(else (cons (car set) (adjoin-set x (cdr set))))

))

* Should run on average O(n/2)



(define (union-set set1 set2)

(cond ((null? set1) set2)

((null? set2) set1)

((= (car set1) (car set2))

(cons (car set1) (union-set (cdr set1) (cdr set2))))

((> (car set1) (car set2))

(cons (car set2) (union-set set1 (cdr set2))))

(else (cons (car set1) (union-set (cdr set1) set2)))))

(define test-set1 (list 1 2 4 8))

(define test-set2 (list 3 4 5 6))

(display (union-set test-set1 test-set2))

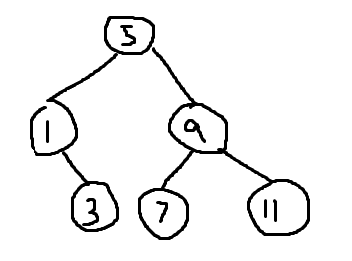
* Run on O(n) because it only compares and cons’s the list in one single operation.



* The two procedures produces the same list if given same tree.
* Figure 2.16 tree to list: (1 3 5 7 9 11), (1 3 7 5 9 11), (1 3 5 9 7 11)
* Tree-list-1 has complexity of O(n log n) because it uses append every time to concatenate a list, which is O(n) combined with traversing half of list to balance the tree of O(log n).
* Tree-list-2 has complexity of O(n) because it traverses every list while concatenating, which will return the final result after it finished.



* Partial-tree split the tree into three:
  + The entry
  + The left tree
    - It first determines what size the left tree should be in order to be balanced
    - Then it returns a tree with the correct elements
  + The right tree
    - It first determines what size the right tree should be in order to be balanced
    - Then it returns a tree with the correct elements
* Then partial-tree combines the entry, left tree and right tree to form a balanced tree



Partial tree from (1 3 5 7 9 11), empty entry is not drawn

* O(n) because it traverses every node to return a balanced tree



(define (union-set set1 set2)

(define (union-set-iter set1 set2)

(cond ((null? set1) set2)

((null? set2) set1)

(else (let ((entry1 (car set1))

(entry2 (car set2)))

(cond

((equal? entry1 entry2)

(cons entry1 (union-set-iter (cdr set1) (cdr set2))))

((< entry1 entry2)

(cons entry1 (union-set-iter (cdr set1) set2)))

((> entry1 entry2)

(cons entry2 (union-set-iter set1 (cdr set2)))))))))

(list->tree (union-set-iter (tree->list set1) (tree->list set2))))

(define (intersection-set set1 set2)

(define (intersection-set-iter set1 set2)

(cond ((null? set1) nil)

((null? set2) nil)

(else (let ((entry1 (car set1))

(entry2 (car set2)))

(cond

((equal? entry1 entry2)

(cons entry1 (intersection-set-iter (cdr set1) (cdr set2))))

((< entry1 entry2)

(intersection-set-iter (cdr set1) set2))

((> entry1 entry2)

(intersection-set-iter set1 (cdr set2))))))))

(list->tree (intersection-set-iter (tree->list set1) (tree->list set2))))

(display (tree->list (intersection-set (list->tree (list 1 7 9)) (list->tree (list 1 2 7)))))  
(display (tree->list (union-set (list->tree (list 1 7 9)) (list->tree (list 1 2 7)))))

* This problem is really hard so I cheated a bit with union-set, but intersection-set is my own thought (although it is heavily based on union-set)
* This solution is O(n) because:
  + It first converts the two sets in the function into list, which is O(2 \* n)
  + The iteration function itself is O(n) because at worst case it iterates every item in the two sets once.
  + Finally, list->tree is O(n)
  + So O(n) + O(n) + (2 \* n) = O(n)



(define (lookup given-key set-of-records)

(cond ((null? set-of-records) #f)

((equal? given-key (key (entry set-of-records)))

#t)

((< given-key (key (entry set-of-records)))

(lookup given-key (left-branch set-of-records)))

((> given-key (key (entry set-of-records)))

(lookup given-key (right-branch set-of-records)))))