```
;;
;; *************************************************
;; Sample Solutions
;; CS 135 Fall 2019
;; Assignment 07, Problem 1
;; *************************************************
;;


(define-struct node (key val left right))
;; A Node is a (make-node Nat Sym BSTD BSTD)
;; requires: key > every key in left BSTD
;;           key < every key in right BSTD

;; A binary search tree dictionary (BSTD) is one of:
;; * empty
;; * Node


(define small (make-node 2 'b
                         (make-node 1 'a empty empty)
                         (make-node 3 'c empty empty)))

(define large (make-node 24 'x
                         (make-node 23 'w empty empty)
                         (make-node 26 'z
                                    (make-node 25 'y empty empty)
                                    empty)))

(define letters (make-node 10 'j small large))


;; 1a)

;;(range-count dict low high) produces the number of keys that are
;;  >= low and < high in dict
;; range-count: BSTD Nat Nat -> Nat
;; requires: low < high
;; Examples:
(check-expect (range-count empty 0 100) 0)
(check-expect (range-count large 20 25) 2)

(define (range-count dict low high)
  (cond
    [(empty? dict) 0]
    [(and (<= low (node-key dict))
```

```
          (< (node-key dict) high))
     (add1 (+ (range-count (node-left dict) low high)
              (range-count (node-right dict) low high)))]
    [(> low (node-key dict))
     (range-count (node-right dict) low high)]
    [(< high (node-key dict))
     (range-count (node-left dict) low high)]
    [else (+ (range-count (node-left dict) low high)
             (range-count (node-right dict) low high))]))

;; Tests:
(check-expect (range-count small 100 125) 0)
(check-expect (range-count small 3 125) 1)
(check-expect (range-count small 4 125) 0)
(check-expect (range-count large 0 1) 0)
(check-expect (range-count large 0 24) 1)
(check-expect (range-count large 0 23) 0)
(check-expect (range-count letters 10 11) 1)
(check-expect (range-count letters 9 10) 0)
(check-expect (range-count letters 9 11) 1)
(check-expect (range-count letters 11 12) 0)
(check-expect (range-count letters 8 9) 0)
(check-expect (range-count letters 4 23) 1)
(check-expect (range-count letters 3 23) 2)
(check-expect (range-count letters 4 24) 2)
(check-expect (range-count letters 1 27) 8)



;;(range-query dict low high) produces a list of the values in dict
;;  whose keys are >= low and < high in ascending order by key
;; range-query dict: BSTD Nat Nat -> (listof Sym)
;; requires: low < high
;; Examples:
(check-expect (range-query empty 0 100) empty)
(check-expect (range-query large 20 25) '(w x))

(define (range-query dict low high)
  (cond
    [(empty? dict) empty]
    [(and (<= low (node-key dict))
          (< (node-key dict) high))
     (append (range-query (node-left dict) low high)
             (cons (node-val dict)
                   (range-query (node-right dict) low high)))]
    [(> low (node-key dict))
```

```
     (range-query (node-right dict) low high)]
    [(< high (node-key dict))
     (range-query (node-left dict) low high)]
    [else (append (range-query (node-left dict) low high)
                  (range-query (node-right dict) low high))]))

;; Tests:
(check-expect (range-query small 100 125) '())
(check-expect (range-query small 3 125) '(c))
(check-expect (range-query small 4 125) '())
(check-expect (range-query large 0 1) '())
(check-expect (range-query large 0 24) '(w))
(check-expect (range-query large 0 23) '())
(check-expect (range-query letters 10 11) '(j))
(check-expect (range-query letters 9 10) '())
(check-expect (range-query letters 9 11) '(j))
(check-expect (range-query letters 11 12) '())
(check-expect (range-query letters 8 9) '())
(check-expect (range-query letters 4 23) '(j))
(check-expect (range-query letters 3 23) '(c j))
(check-expect (range-query letters 4 24) '(j w))
(check-expect (range-query letters 1 27) '(a b c j w x y z))



;;
;; *************************************************
;; Sample Solutions
;; CS 135 Fall 2019
;; Assignment 07, Problem 2
;; *************************************************
;;


(define-struct bnode (val left right))
;; A BNode is a (make-bnode Nat ITree ITree)
;; requires: val > every val in left ITree
;;           val < every val in right ITree


;; An ITree (Inteveral Tree) is one of:
;; * a Sym (a leaf)
;; * a BNode (an boundary node)


(define small-itree (make-bnode 5 'a 'b))
```

```
(define large-itree (make-bnode 25 (make-bnode 23 'x 'y) 'z))

(define letters-itree (make-bnode 10 small-itree large-itree))


;;(it-lookup it n) produces the symbol from it that is
;;  associated with the interval that contains n
;; it-lookup: ITree Nat -> Sym
;; Examples:
(check-expect (it-lookup 'a 5) 'a)
(check-expect (it-lookup letters-itree 8) 'b)

(define (it-lookup it n)
  (cond
    [(symbol? it) it]
    [(< n (bnode-val it))
     (it-lookup (bnode-left it) n)]
    [else (it-lookup (bnode-right it) n)]))

;; Tests:
(check-expect (it-lookup small-itree 4) 'a)
(check-expect (it-lookup small-itree 5) 'b)
(check-expect (it-lookup small-itree 6) 'b)
(check-expect (it-lookup letters-itree 0) 'a)
(check-expect (it-lookup letters-itree 7) 'b)
(check-expect (it-lookup letters-itree 12) 'x)
(check-expect (it-lookup letters-itree 23) 'y)
(check-expect (it-lookup letters-itree 25) 'z)



;;
;; *************************************************
;; Sample Solutions
;; CS 135 Fall 2019
;; Assignment 07, Problem 3
;; *************************************************
;;


;; An RTree (Rectangle Tree) is one of:
;; * Sym (a leaf)
;; * XNode (a boundary node for an x value)
;; * YNode (a boundary node for a y value)
```

```
(define-struct xnode (val left right))
;; An XNode is a (make-xnode Nat RTree RTree)
;; requires: val > every xnode-val in left RTree
;;           val < every val in right RTree

(define-struct ynode (val below above))
;; A YNode is a (make-ynode Nat RTree RTree)
;; requires: val > every ynode-val in below RTree
;;           val < every ynode-val in above RTree


(define small-x1 (make-xnode 6 'a 'b))
(define small-x2 (make-xnode 4 'c 'd))
(define small-x3 (make-xnode 10 'e 'f))
(define small-y (make-ynode 8 'e 'f))

(define med-y (make-ynode 10 small-x1 small-x2))
(define large-x (make-xnode 9 med-y small-x3))


;; 3a)

;; rtree-template: RTree -> Any
(define (rtree-template rt)
  (cond
    [(symbol? rt) (... rt ... )]
    [(xnode? rt) (xnode-template rt)]
    [(ynode? rt) (ynode-template rt)]))


;; xnode-template: XNode -> Any
(define (xnode-template node)
  (... (xnode-val node) ...
       (rtree-template (xnode-left node)) ...
       (rtree-template (xnode-right node)) ...))


;; ynode-template: YNode -> Any
(define (ynode-template node)
  (... (ynode-val node) ...
       (rtree-template (ynode-below node)) ...
       (rtree-template (ynode-above node)) ...))


;; 3b)
```

```
;;(rt-lookup rt pos) produces the symbol label of the region
;;  in rt containing pos
;; rt-lookup: RTree Posn -> Sym
;; Examples:
(check-expect (rt-lookup 'a (make-posn 3 4)) 'a)
(check-expect (rt-lookup med-y (make-posn 2 8)) 'a)
(check-expect (rt-lookup small-x1 (make-posn 5 0)) 'a)

(define (rt-lookup rt pos)
  (cond
    [(symbol? rt) rt]
    [(and (xnode? rt)
          (< (posn-x pos) (xnode-val rt)))
     (rt-lookup (xnode-left rt) pos)]
    [(xnode? rt) (rt-lookup (xnode-right rt) pos)]
    [(< (posn-y pos) (ynode-val rt))
     (rt-lookup (ynode-below rt) pos)]
    [else (rt-lookup (ynode-above rt) pos)]))

;; Tests:
(check-expect (rt-lookup small-y (make-posn 0 8)) 'f)
(check-expect (rt-lookup small-y (make-posn 0 7)) 'e)
(check-expect (rt-lookup small-y (make-posn 0 9)) 'f)
(check-expect (rt-lookup small-x1 (make-posn 6 0)) 'b)
(check-expect (rt-lookup small-x1 (make-posn 7 0)) 'b)


;; 3c)

;;(max-result res1 res2) produces the largest number among res1 and res2
;; max-result: (anyof Nat Sym) (anyof Nat Sym) -> (anyof Nat Sym)
;; Examples:
(check-expect (max-result 4 5) 5)
(check-expect (max-result 'a 5) 5)

(define (max-result res1 res2)
  (cond
    [(symbol? res1) res2]
    [(symbol? res2) res1]
    [else (max res1 res2)]))

;; Tests:
(check-expect (max-result 4 'b) 4)
(check-expect (max-result 'a 'b) 'b)
```

```
;;(rt-max-x rt) produces the largest x boundary in rt
;; rt-max-x: RTree -> (anyof Nat 'None)
;; Examples:
(check-expect (rt-max-x 'a) 'None)
(check-expect (rt-max-x large-x) 10)
(check-expect (rt-max-x med-y) 6)

(define (rt-max-x rt)
  (cond
    [(symbol? rt) 'None]
    [(ynode? rt) (max-result (rt-max-x (ynode-above rt))
                             (rt-max-x (ynode-below rt)))]
    [else (max-result (xnode-val rt)
                      (rt-max-x (xnode-right rt)))]))

;; Tests:
(check-expect (rt-max-x (make-ynode 10 small-x2 small-x1)) 6)
(check-expect (rt-max-x small-x1) 6)


(define-struct win (wid rtree))
;; A Win (Window) is a (make-win Nat RTree)
;; requires: points outside the window are labelled 'None

(define win1 (make-win 1 (make-xnode 2 'None
                                     (make-xnode 8 (make-ynode 1 'None
                                                               (make-ynode
4 'win1 'None)) 'None))))
(define win2 (make-win 2 (make-xnode 4 'None
                                     (make-xnode 10 (make-ynode 3 'None
                                                                (make-ynode
6 'win2 'None)) 'None))))
(define win3 (make-win 3 (make-xnode 5 'None
                                     (make-xnode 11 (make-ynode 0 'None
                                                                (make-ynode
3 'win3 'None)) 'None))))


;; 3d)

;;(shift-rtree rt x y) shifts rt by (x,y)
;; shift-rtree: RTree Int Int -> RTree
;; Examples/Tests: See move-win
(define (shift-rtree rt x y)
  (cond
    [(symbol? rt) rt]
```

```
    [(ynode? rt) (make-ynode (+ y (ynode-val rt))
                             (shift-rtree (ynode-below rt) x y)
                             (shift-rtree (ynode-above rt) x y))]
    [else (make-xnode (+ x (xnode-val rt))
                      (shift-rtree (xnode-left rt) x y)
                      (shift-rtree (xnode-right rt) x y))]))


;;(move-win wi x y) moves win by (x,y)
;; move-win: Win Int Int -> Win
;; Examples:
(check-expect (move-win win1 2 2)
              (make-win 1
                        (make-xnode 4 'None
                                    (make-xnode 10 (make-ynode 3 'None
                                                               (make-ynode
6 'win1 'None)) 'None))))
(check-expect (move-win win2 1 -3)
              (make-win 2 (make-xnode 5 'None
                                    (make-xnode 11 (make-ynode 0 'None
                                                               (make-ynode
3 'win2 'None)) 'None))))

(define (move-win wi x y)
  (make-win (win-wid wi) (shift-rtree (win-rtree wi) x y)))

;; Test:
(check-expect (move-win win2 -2 -2)
              (make-win 2 (make-xnode 2 'None
                                    (make-xnode 8 (make-ynode 1 'None
                                                               (make-ynode
4 'win2 'None)) 'None))))


;; A WinSys is one of:
;; * empty
;; * (cons Win WinSys)


;; 3e)

;;(winsys-lookup ws pos) produces the top window in ws that contains pos
;; winsys-lookup: WinSys Posn -> (list Nat Sym)
;; Examples:
(check-expect (winsys-lookup empty (make-posn 1 1)) (list 0 'None))
(check-expect (winsys-lookup (list win1 win2 win3) (make-posn 7 2)) (list
```

```
1 'win1))

(define (winsys-lookup ws pos)
  (cond
    [(empty? ws) (list 0 'None)]
    [(symbol=? 'None (rt-lookup (win-rtree (first ws)) pos))
     (winsys-lookup (rest ws) pos)]
    [else (list (win-wid (first ws)) (rt-lookup (win-rtree (first ws))
pos))]))

;; Tests:
(check-expect (winsys-lookup (list win1 win2 win3) (make-posn 100 100))
(list 0 'None))
(check-expect (winsys-lookup (list win1 win2 win3) (make-posn 9 5)) (list
2 'win2))
```