```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; CS 135 Winter 2014 - Assignment 6 Q1 - Solution
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-struct node (key value left right))
    ;; A binary search tree (BST) is one of
    ;; * empty
    ;; * a (make-node Num String Bst Bst),
    ;; where for each node (make-node k v l r), every node in the subtree
    ;; rooted at l has a key less than k, and every node in the subtree rooted
    ;; at r has a key greater than k.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; More Constants for examples & tests
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define tree0 empty)
(define tree1 (make-node 5 "5" empty empty))
(define tree2 (make-node 5 "5" (make-node 3 "3" empty empty) empty))
; 5
; └3
(define tree3a (make-node 5 "5" (make-node 3 "3" empty empty)
                          (make-node 7 "7" empty empty)))
;  ┌7
; 5
; └3
(define tree3b (make-node 3 "3" empty
                          (make-node 5 "5" empty
                                     (make-node 7 "7" empty empty))))
;    ┌7
;  ┌5
; 3
(define tree3c (make-node 7 "7"
                          (make-node 5 "5"
                                     (make-node 3 "3" empty empty)
                                     empty) empty))
; 7
; └5
;   └3
(define tree3d (make-node 7 "7"
                          (make-node 3 "3" empty
                                     (make-node 5 "5" empty empty))
                          empty))
; 7
; │ ┌5
; └3

(define tree-2 (make-node 2 "b" empty empty))
(define tree-12 (make-node 2 "b" (make-node 1 "a" empty empty) empty))
(define tree-23 (make-node 2 "b" empty (make-node 3 "c" empty empty)))
(define tree-123 (make-node 2 "b" (make-node 1 "a" empty empty)
                                  (make-node 3 "c" empty empty)))

(define sample-bst-smallnum
    (make-node 8 "8"
    (make-node 4 "4"
    (make-node 2 "2"
    (make-node 1 "1" empty empty)
    (make-node 3 "3" empty empty))
    (make-node 6 "6"
    (make-node 5 "5" empty empty)
    (make-node 7 "7" empty empty)))
    (make-node 12 "12"
    (make-node 10 "10"
    (make-node 9 "9" empty empty)
    (make-node 11 "11" empty empty))
    (make-node 14 "14"
    (make-node 13 "13" empty empty) empty))))
  (define sample-bst-bignum
    (make-node 154 "150"
    (make-node 110 "110" empty
    (make-node 142 "142"
    (make-node 111 "111" empty empty)
    (make-node 144 "144" empty empty)))
    (make-node 212 "212"
    (make-node 177 "177" empty empty)
    (make-node 242 "242" empty
    (make-node 266 "266" empty
    (make-node 391 "391"
    (make-node 305 "305" empty empty) empty))))))
(define sample-bst-new (make-node 75 "75" sample-bst-smallnum sample-bst-bignum))
(define sample-full-bst-new
  (make-node 2 "b" (make-node 1 "a" empty empty)
             (make-node 4 "d" (make-node 3 "c" empty empty)
                        (make-node 6 "f" (make-node 5 "e" empty empty)
                                   (make-node 8 "h" (make-node 7 "g" empty empty)
                                              (make-node 9 "i" empty empty))))))
(define another-full-bst
  (make-node 2 "b" (make-node 1 "a" empty empty)
             (make-node 4 "d" (make-node 3 "c" empty empty)
                        (make-node 6 "f" (make-node 5 "e" empty empty)
                                   (make-node 8 "h" (make-node 7 "g" empty empty)
                                              (make-node 9 "i" empty empty))))))

(define sample-perfect-bst-new
  (make-node 2 "b" (make-node 1 "a" (make-node 0 "c" empty empty) (make-node 1.5
"aa" empty empty))
             (make-node 4 "d" (make-node 3 "c" empty empty) (make-node 6 "f" empty
empty))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1 (a)
;; bst-add: Bst Num String -> Bst
;; Purpose: Adds the given key and value to the given BST.
;; Examples:
(check-expect (bst-add tree0 5 "five") (make-node 5 "five" empty empty))
(check-expect (bst-add tree1 6 "six") (make-node 5 "5" empty (make-node 6 "six"
empty empty)))
(check-expect (bst-add tree1 5 "55") (make-node 5 "55" empty empty))
;; Definition:
(define (bst-add abst k v)
  (cond
    [(empty? abst) (make-node k v empty empty)]
    [(= k (node-key abst)) (make-node k v (node-left abst) (node-right abst))]
    [(< k (node-key abst)) (make-node (node-key abst)
                                      (node-value abst)
                                      (bst-add (node-left abst) k v)
                                      (node-right abst))]
    [else (make-node (node-key abst)
                       (node-value abst)
                       (node-left abst)
                       (bst-add (node-right abst) k v))]))
;; Tests
(check-expect (bst-add tree3a 5 "55")
              (make-node 5 "55" (make-node 3 "3" empty empty)
                         (make-node 7 "7" empty empty)))
(check-expect (bst-add tree3a 4 "44")
              (make-node 5 "5" (make-node 3 "3" empty
                                          (make-node 4 "44" empty empty))
                         (make-node 7 "7" empty empty)))
(check-expect (bst-add tree3a 6 "66")
              (make-node 5 "5" (make-node 3 "3" empty empty)
                         (make-node 7 "7" (make-node 6 "66" empty empty) empty)))
(check-expect (bst-add tree3a 3 "33")
              (make-node 5 "5" (make-node 3 "33" empty empty)
                         (make-node 7 "7" empty empty)))
(check-expect (bst-add tree3a 2 "2")
              (make-node 5 "5" (make-node 3 "3" (make-node 2 "2" empty empty) empty)
                         (make-node 7 "7" empty empty)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1 (b)

;; bst-full? : Bst -> Boolean
;; Purpose: Produces true iff the given BST is full.
;; Examples:
(check-expect (bst-full? sample-bst-smallnum) false)
(check-expect (bst-full? sample-bst-bignum) false)
(check-expect (bst-full? tree1) true)
;; Definition:
(define (bst-full? abst)
  (cond
    [(empty? abst) true]
    [(and (empty? (node-left abst)) (empty? (node-right abst))) true]
    [(or (empty? (node-left abst)) (empty? (node-right abst))) false]
    [else (and (bst-full? (node-left abst)) (bst-full? (node-right abst)))]))
;; Tests:
(check-expect (bst-full? tree0) true)
(check-expect (bst-full? tree1) true)
(check-expect (bst-full? tree2) false)
(check-expect (bst-full? tree3a) true)
(check-expect (bst-full? tree3b) false)
(check-expect (bst-full? tree3c) false)
(check-expect (bst-full? tree3d) false)
(check-expect (bst-full? sample-bst-new) false)
(check-expect (bst-full? sample-full-bst-new) true)
(check-expect (bst-full? another-full-bst) true)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1 (c)
;; bst-height : Bst -> Nat
;; Purpose: Produces the height of the given BST
;; Examples:
  (check-expect (bst-height empty) 0)
  (check-expect (bst-height tree-123) 2)
;; Definition:
(define (bst-height abst)
  (cond
    [(empty? abst) 0]
    [else (add1 (max (bst-height (node-left abst)) (bst-height (node-right
abst))))]))
;; Tests:
  (check-expect (bst-height empty) 0)
  (check-expect (bst-height tree-2) 1)
  (check-expect (bst-height tree-12) 2)
  (check-expect (bst-height tree-123) 2)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1 (d)
;; bst-perfect? : Bst -> Boolean
;; Purpose: Produces true iff the given BST is perfect
;; Examples:
  (check-expect (bst-perfect? empty) true)
  (check-expect (bst-perfect? tree-2) true)
;; Definition
(define (bst-perfect? abst)
  (cond
    [(empty? abst) true]
    [else (and (bst-perfect? (node-left abst)) (bst-perfect? (node-right abst))
               (= (bst-height (node-left abst)) (bst-height (node-right abst))))]))
;; Tests:
  (check-expect (bst-perfect? empty) true)
  (check-expect (bst-perfect? tree-12) false)
  (check-expect (bst-perfect? tree-23) false)
  (check-expect (bst-perfect? tree-123) true)
  (check-expect (bst-perfect? sample-perfect-bst-new) true)
  (check-expect (bst-perfect? sample-full-bst-new) false)
  (check-expect (bst-perfect? another-full-bst) false)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1 (e)
;; bst->sal: Bst -> (listof (list Num String))
;; or
;; bst->sal: Bst -> AL
;; Purpose: Produces a sorted Association List (AL) from a given BST
;; Examples:
(check-expect (bst->sal empty) empty)
(check-expect (bst->sal sample-bst-smallnum)
              '((1 "1") (2 "2") (3 "3") (4 "4") (5 "5") (6 "6")
                          (7 "7") (8 "8") (9 "9") (10 "10") (11 "11")
                          (12 "12") (13 "13") (14 "14")))
;; Definition:
(define (bst->sal tree)
  (cond
    [(empty? tree) empty]
    [else (append
           (bst->sal (node-left tree))
           (list (list (node-key tree) (node-value tree)))
           (bst->sal (node-right tree)))]))
;; Tests:
(check-expect (bst->sal tree0) empty)
(check-expect (bst->sal tree1) '((5 "5")))
(check-expect (bst->sal tree2) '((3 "3")(5 "5")))
(check-expect (bst->sal tree3a) '((3 "3")(5 "5")(7 "7")))
(check-expect (bst->sal tree3b) '((3 "3")(5 "5")(7 "7")))
```

```
(check-expect (bst->sal tree3c) '((3 "3")(5 "5")(7 "7")))
(check-expect (bst->sal tree3d) '((3 "3")(5 "5")(7 "7")))


;; Data definition for Q2 and Q3
;; A leaf-labelled tree (Llt) is one of the following:
;; * empty
;; * (cons l1 l2), where l1 is a non-empty Llt and l2 is an Llt
;; * (cons v l), where v is an Int and l is an Llt

;; Template for Q2 and Q3
;; (define (my-llt-fn tree)
;;   (cond
;;     [(empty? tree) ...]
;;     [(cons? (first tree))
;;        ... (my-llt-fn (first tree)) ...
;;        ... (my-llt-fn (rest tree))) ... ]
;;     [(integer? (first tree)) ... (first tree) ... (my-llt-fn (rest tree))]))


;; Question 2:
;; height: Llt -> Nat
;; Purpose:
;; Produce the height of leaf-labelled tree t
;; Examples:
;; (height empty) => 0
;; (height (list 1 2 3)) => 1
;; (height (list (list 1) (list (list 7 9)))) => 3
(define (height t)
  (cond
    [(empty? t) 0]
    [(cons? (first t))
     (max (add1 (height (first t)))
          (height (rest t)))]
    [else (max 1 (height (rest t)))]))
;; Tests:
(check-expect (height empty) 0)
(check-expect (height (list 1 2 3)) 1)
(check-expect (height '(1 ((7 9)))) 3)
(check-expect (height '(((7 9) 1 (4 5)))) 3)
(check-expect (height '(1 (2) 3)) 2)
(check-expect (height '((((1))))) 4)


;; Question 3:

;; number-leaves: Llt -> Nat
;; Purpose:
;; Produce the number of leaves in leaf-labelled tree t
;; at this particular depth
;; Examples:
;; (number-leaves empty) => 0
;; (number-leaves (list 1 2 (list 3) 4 (list 5))) => 3
;; (number-leaves (list (list 1 2 3))) => 0
(define (number-leaves t)
  (cond
    [(empty? t) 0]
    [(cons? (first t)) (number-leaves (rest t))]
    [else (+ 1 (number-leaves (rest t)))]))
;; Tests:
(check-expect (number-leaves empty) 0)
(check-expect (number-leaves (list 1 2 (list 3) 4 (list 5))) 3)
(check-expect (number-leaves (list (list 1 2 3))) 0)

;; num-siblings: Llt Int -> (union Nat false)
;; Conditions:
;;  PRE:  all leaves are distinct (i.e., no duplicate leaves)
;; Purpose:  produce the number of leaves that are siblings
;; of n in tree t, or else produce false if n is not in t
;; Examples:
;; (num-siblings empty 5) => false
;; (num-siblings '(3) 3) => 0
;; (num-siblings '(5 (3 1 2) 4) 3) => 2
(define (num-siblings t n)
  (cond
    [(empty? t) false]
    [(cons? (first t))
     (cond
       [(member? n (first t))
        (sub1 (number-leaves (first t)))]
       [(integer? (num-siblings (first t) n))
        (num-siblings (first t) n)]
       [else (num-siblings (rest t) n)])]
    [else
     (cond
       [(member? n t) (sub1 (number-leaves t))]
       [else (num-siblings (rest t) n)])]))
;; Tests
(check-expect (num-siblings empty 3) false)
(check-expect (num-siblings '(3) 3) 0)
(check-expect (num-siblings '(3 1 2) 3) 2)
(check-expect (num-siblings '((3 1 2) 5) 3) 2)
(check-expect (num-siblings '(3 1 2) 3) 2)
(check-expect (num-siblings '(2 1 3 (8) 7) 3) 3)
(check-expect (num-siblings '(2 1 3 (8) 7) 8) 0)
(check-expect (num-siblings '(5 (6) (2 1 3 (8) 7)) 3) 3)
(check-expect (num-siblings '(5 (6) (2 1 3 (8) 7)) 6) 0)
(check-expect (num-siblings '(5 (6) (2 1 3 (8) 7)) 8) 0)
(check-expect (num-siblings '(5 (6) (2 1 3 (8) 7)) 7) 3)
(check-expect (num-siblings '((6) (2 1 3 (8) 7)) 3) 3)
(check-expect (num-siblings '((6) (2 1 3 (8) 7)) 9) false)


;; Question 4:

;; A boolean expression (Bexp) is either
;; * a Boolean value (true or false),
;; * a comparison expression (Cexp), or
;; * a compound boolean expression (Cbexp)

;; Template
;; my-bexp-fn: Bexp -> Any
;; (define (my-bexp-fn bexpr)
;;   (cond
;;     [(boolean? bexpr) ...]
;;     [(comp-exp? bexpr) ... (my-comp-expr-fn bexpr) ...]
;;      [(compoundb-exp? bexpr) ... (my-compoundb-exp bexpr) ...]))

(define-struct comp-exp (fn arg1 arg2))
;; A comparison expression (Cexp) is a structure
;; (make-comp-exp f a1 a2), where
;; * f is a Symbol from the set '>, '<, and '=
;; * a1 is a Num
;; * a2 is a Num

;; Template
;; my-comp-expr-fn: Cexp -> Any
;; (define (my-comp-expr-fn cexpr)
;;  ... (comp-exp-fn cexpr)
;;  ... (comp-exp-arg1 expr)
;;  ... (comp-exp-arg2 expr))

(define-struct compoundb-exp (op args))
;; A compound boolean expression (Cbexp) is a structure
;; (make-compoundb-exp b alist), where
;; * b is a Symbol from the set 'and and 'or
;; * alist is a Cbexplist

;; Template
;; my-compoundb-exp-fn: Cbexp -> Any
;; (define (my-compoundb-exp-fn cbexpr)
;;   ... (compoundb-exp-op cbexpr)
;;   ... (my-cbexplist-fn (compoundb-exp-args cbexpr)))

;; A Cbexplist is either
;; * empty, or
;; * (cons e alist), where e is a Bexp and alist is a Cbexplist

;; Template
;; my-cbexplist-fn: Cbexplist -> Any
;; (define (my-cbexplist-fn cbexprlst)
;;   (cond
;;     [(empty? cbexprlst) ...]
;;     [else
;;       ... (my-bexp-fn (first cbexprlst))
;;       ... (my-cbexplist-fn (rest cbexprlst))]))

;; START OF MAIN FUNCTION

;; bool-eval: Bexp -> Boolean
;; Purpose: given a boolean expression bexpr,
;; produce the boolean value of the expression
;; simplified
;; Examples:
;; (bool-eval true) => true
;; (bool-eval (make-comp-exp '> 6 9)) => false
;; (bool-eval (make-compoundb-exp 'or
;;              (list (make-comp-exp '> 5 7)
;;                    false (make-comp-exp '< 2 3)))) => true
(define (bool-eval bexpr)
  (cond
    [(boolean? bexpr) bexpr]
    [(comp-exp? bexpr) (bool-eval-cexp bexpr)]
    [(compoundb-exp? bexpr) (bool-eval-cbexp bexpr)]))
```

```
;; Tests:
(check-expect (bool-eval true) true)
(check-expect (bool-eval false) false)
(check-expect (bool-eval (make-comp-exp '> 6 9)) false)
(check-expect (bool-eval (make-comp-exp '> 9 6)) true)
(check-expect (bool-eval (make-comp-exp '= 7 5)) false)
(check-expect (bool-eval (make-comp-exp '= 5 5)) true)
(check-expect (bool-eval
               (make-compoundb-exp
                'or
                (list (make-comp-exp '> 5 7)
                      false (make-comp-exp '< 2 3)))) true)
(define be1 (make-compoundb-exp 'and (list true (make-comp-exp '> 5 3) true)))
(define be2 (make-compoundb-exp 'or (list false false (make-comp-exp '= 4 4))))
(define be3 (make-compoundb-exp 'and (list be1 be2 be1)))
(check-expect (bool-eval be3) true)
(check-expect (bool-eval be1) true)
(check-expect (bool-eval be2) true)
(check-expect (bool-eval
               (make-compoundb-exp 'and (list be1 be1 be2 be3 be1 false)))
              false)
(check-expect (bool-eval
               (make-compoundb-exp 'or (list be1 be1 be2 be3 be1 false)))
              true)
(check-expect (bool-eval
               (make-compoundb-exp 'and
                                   (list be1 (make-compoundb-exp
                                              'or (list false false be2)))))
              true)
(check-expect (bool-eval
               (make-compoundb-exp 'and
                                   (list be1 (make-compoundb-exp
                                              'or (list false false be2)))))
              true)


;; bool-eval-cexp: cexp -> boolean
;; Purpose: given a comparison expression cexpr, determine
;; its boolean value when simplified
;; Examples:
;; (bool-eval-cexp (make-comp-exp '= 4 5)) => false
;; (bool-eval-cexp (make-comp-exp '< 4 5)) => true
(define (bool-eval-cexp cexpr)
  (cond
    [(symbol=? (comp-exp-fn cexpr) '=)
     (= (comp-exp-arg1 cexpr) (comp-exp-arg2 cexpr))]
    [(symbol=? (comp-exp-fn cexpr) '>)
     (> (comp-exp-arg1 cexpr) (comp-exp-arg2 cexpr))]
    [(symbol=? (comp-exp-fn cexpr) '<)
     (< (comp-exp-arg1 cexpr) (comp-exp-arg2 cexpr))]))
;; Tests:
(check-expect (bool-eval-cexp (make-comp-exp '= 4 5)) false)
(check-expect (bool-eval-cexp (make-comp-exp '= 5 5)) true)
(check-expect (bool-eval-cexp (make-comp-exp '< 4 5)) true)
(check-expect (bool-eval-cexp (make-comp-exp '< 5 4)) false)
(check-expect (bool-eval-cexp (make-comp-exp '> 4 5)) false)
(check-expect (bool-eval-cexp (make-comp-exp '> 5 4)) true)
```

```
;; bool-eval-cbexp: cbexp -> boolean
;; Purpose:  given a compound boolean expression cbexpr,
;; produce the boolean value when cbexpr is simplified
;; Examples:
;; (bool-eval-cbexp
;; (make-compoundb-exp 'and (list true (make-comp-exp '> 5 3) true)))
;; => true
;; (bool-eval-cbexp
;; (make-compoundb-exp 'or (list false (make-comp-exp '< 5 3) false)))
;; => false
(define (bool-eval-cbexp cbexpr)
  (bool-eval-cbexplist (compoundb-exp-op cbexpr)
                       (compoundb-exp-args cbexpr)))
;; Tests:
(check-expect (bool-eval-cbexp
               (make-compoundb-exp
                'and (list true (make-comp-exp '> 5 3) true)))
              true)
(check-expect (bool-eval-cbexp
               (make-compoundb-exp 'or (list false (make-comp-exp '< 5 3) false)))
              false)


;; bool-eval-cbexplist: symbol cbexplist -> boolean
;; Purpose:  Evaluate the given cbexplist cbexprlist to
;; its boolean value
;; Examples:
;; (bool-eval-cbexplist 'and (list true true true)) => true
;; (bool-eval-cbexplist 'or (list false true false)) => true
(define (bool-eval-cbexplist op arglist)
  (cond
    [(empty? arglist) (symbol=? op 'and)]
    [(symbol=? op 'and)
     (and (bool-eval (first arglist))
          (bool-eval-cbexplist op (rest arglist)))]
    [else
     (or (bool-eval (first arglist))
         (bool-eval-cbexplist op (rest arglist)))]))
;; Tests:
(check-expect (bool-eval-cbexplist 'and (list true true true))
              true)
(check-expect (bool-eval-cbexplist 'or (list false true false))
              true)
(check-expect (bool-eval-cbexplist 'or (list false false false))
              false)
(check-expect (bool-eval-cbexplist 'and (list false true true))
              false)
(check-expect (bool-eval-cbexplist
               'and
               (list true (make-comp-exp '> 5 3) true))
              true)
(check-expect (bool-eval-cbexplist
               'and
               (list false (make-comp-exp '> 5 3) false))
              false)
```